TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Informaatikainstituut
Infosüsteemide õppetool

Reapõhise ja veerupõhise andmete salvestamise võrdlus kahe SQL-andmebaasisüsteemi näitel

Magistritöö

Üliõpilane: Siim Puustusmaa
Üliõpilaskood: 132386
Juhendaja: Erki Eessaar

Tallinn
2016

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

---------- ----------

(*kuupäev*) (*allkiri*)

Reapõhise ja veerupõhise andmete salvestamise võrdlus kahe SQL-andmebaasisüsteemi näitel

Annotatsioon

Käesolevas töös uuritakse, kuidas mõjutab veerupõhine ja reapõhine andmete salvestamine SQL-andmebaasisüsteemides andmekäitluse operatsioonide jõudlust ning nendes süsteemides loodud andmebaaside andmemahte. Töö alguses püstitatakse kirjanduse alusel hüpoteesid rea- ja veerupõhise andmete salvestamise kohta üldiselt, mille kehtivusele antakse töö lõpuks konkreetsete süsteemide ja praktiliste katsete põhjal hinnang. Andmebaasisüsteeme võrreldakse omavahel päringute töökiiruse ja andmete haldamise (sisestamine, muutmine, kustutamine) operatsioonide jõudluse seisukohalt. Rea- ja veerupõhine salvestamine tähendab antud juhul andmete erinevat ning lõppkasutaja eest peidetud organiseerimist andmebaasisüsteemi sisemisel tasemel. Kontseptuaalsel tasemel töötavad kasutajad endiselt SQL tabelitega. Töö käigus uuritakse, millist tüüpi operatsioonid saavad kasu andmete veerupõhisest salvestamisest. Töö eksperimendi osas võetakse vaatluse alla kaks SQL-andmebaasisüsteemi, milleks on Microsoft SQL Server 2014 ja MonetDB 5. Microsoft SQL Server realiseerib andmete reapõhise salvestamise. Samas on selles realiseeritud indeksi tüüp, mis salvestab andmed veerupõhiselt. MonetDB esindab aga puhast veerupõhise salvestamisega andmebaasisüsteemi. Nendes andmebaasisüsteemides luuakse andmebaasid, mis kõik täidetakse samade andmetega ning milles viiakse läbi erinevad katsed.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 114 leheküljel, 14 peatükki, 41 joonist ja 12 tabelit.

A Comparison of Row Storage and Columnar Storage in the Example of Two SQL Database Management Systems

Abstract

In this thesis a research will be conducted about how the use of columnar storage and row storage in SQL Database Management Systems (DBMSs) influences the performance of data management operations as well as the size of the databases created in those systems. At the beginning of the thesis, we outline some general hypothesis based on literature about row storage and columnar storage. These hypotheses will be evaluated at the end of the thesis based on the particular DBMSs and practical experiments. We compare the DBMSs by measuring the execution time of different queries and data management (insert, update, and delete) operations. Columnar storage and row storage in this case means how the data is organized at the internal level of a database. This organization is hidden from database users, who still work at the conceptual level of the database with SQL tables. In the course of this work there will be an investigation to find out the types of operations that profit from the columnar storage. Experiments will be conducted based on two SQL DBMSs – Microsoft SQL Server 2014 and MonetDB 5. Microsoft SQL Server is a DBMS that implements row storage. However, it also supports index type that stores data in the columnar manner. On the other hand, MonetDB is a pure columnar storage DBMS. All databases will be created in those DBMSs and will be filled with identical data. All the tests will be carried out in those databases.

The paper is written in Estonian and contains 114 pages of text, 14 chapters, 41 drawings and 12 tables.

Lühendite ja mõistete sõnastik

BAT ***Binary Association Table***

MonetDB andmebaasisüsteemis sisemiselt kasutatav andmestruktuur. Terviklik SQL tabel fragmenteeritakse (killustatakse) vertikaalselt mitmeks BAT tabeliks.

BLOB ***Binary Large Object***

Andmetüüp, mida andmebaasisüsteemid kasutavad binaarsete andmete salvestamiseks.

CSI ***Colum-Store Index***

Microsoft SQL Server andmebaasisüsteemis olev indeksi tüüp, mis võimaldab tabeli read salvestada veerupõhiselt.

DSM ***Decomposition Storage Model***

Ploki (lehekülje) formaat veerupõhise andmete salvestamise korral.

GB ***Gigabyte***

Andmemahtude mõõtühik. On nii välja kujunenud, et erinevates kontekstides tähistab see mõõtühik erinevaid suurused. Antud töö kontekstis tähendab see vastavalt levinud praktikale 1024^3=1 073 741 824 baiti.

Klasterdamine ***Clustering***

Lähestikku paigutamine, kokku koondamine. Mingi tingimuse alusel sarnased andmed on salvestusstruktuurides füüsiliselt lähestikku, et kiirendada nende andmete üheskoos lugemist.

Kodeerimine ***Encoding***

Andmete viimine vajalikule (standardsele) kujule nende salvestamiseks või töötlemiseks. Kodeeritud andmed on inimesele sageli loetamatud, kuid kodeerimist tundev süsteem saab neid andmeid lugeda ja töödelda ka kodeeritud kujul. Erinevate kodeeringute puhul võib tulemuseks olla erinev andmemaht.

Kokkuvõtte-funktsioon ***Aggregate function***

Andmebaasikeeles (nt SQL) olev funktsioon, mille sisendiks on hulk väärtuseid ja mis tagastab ainult ühe (skalaarse) väärtuse. Tuntumad funktsioonid on `AVG, COUNT, MAX, MIN, SUM`.

Lehekülg	***Page*** Virtuaalmälusüsteemides kujutab lehekülg endast kindlat baitide arvu, mida operatsioonisüsteem käsitleb ühe tervikuna. Operatsioonisüsteemi poolt mällu loetud plokki kutsutakse leheküljeks.
NSM	***N-Ary Storage*** Ploki (lehekülje) formaat reapõhise andmete salvestamise korral. Tuntakse ka nime all *Slotted Pages*.
NTFS	***New Technology File System*** Microsoft Windows NT ja selle järeltulijate failisüsteem. Selles failisüsteemis on ploki (kasutatakse ka terminit klaster) suuruseks vaikimisi 4KB, kuid toetab ka ploki suurusi kuni 64KB'ni.
OID	***Object Identifier*** Väärtus, mida kasutatakse konkreetse tarkvaralise objekti identifitseerimiseks.
OLAP	***Online Analytical Processing*** Suure hulga mitmemõõtmeliste vaadete ja hierarhiatena organiseeritud andmete töötlemise protsess eesmärgiga saada ettevõttele kasulikke statistilisi andmeid või ennustada ette sündmusi.
OLTP	***Online Transaction Processing*** Onlain tehingutöötluse protsess, mille sisuks on üksikute olemite kohta käivate faktide otsimine, lisamine, muutmine ja kustutamine. Mitu sellist operatsiooni võib koondada ka üheks loogiliseks tervikuks e tehinguks e transaktsiooniks.
Pakkimine	***Comperssion*** Andmete viimine vormingusse, mille ainuke eesmärk on andmemahu vähendamine. Pakitud andmete lugemiseks ja töötlemiseks tuleb need kõigepealt lahti pakkida.
Plokk	***Block*** Rühm baite kõvakettal, mida loetakse ja salvestatakse ühe tervikliku üksusena.
Protsessori vahemälu möödalask	***CPU Cache miss*** Olukord, kus järgmise protsessori operatsiooni täitmiseks vajalikke andmeid ei leita protsessori vahemälu hierarhiast (L1, L2, L3) ja seega tuleb pöörduda muutmälu (RAM) poole, mis aga on oluliselt aeglasem kui vahemälu.
Puhver (Microsoft	***Buffer pool*** Microsoft SQL Serveri poolt kasutatav muutmälu piirkond, kuhu salvestatakse

SQL Serveris)	päringutes kasutatud leheküljed.
RLE	***Run-length encoding*** Andmete kodeerimise meetod, kus järjestikused identsed väärtused asendatakse paariga {väärtus, esinemise arv}.
RS	***Read-optimized Store*** *C-Store* tarkvarakomponent, mis pakub kiiret andmete lugemist.
SQL	***Structured Query Language*** Populaarne andmebaasikeel, mis pakub ühe võimaliku relatsioonilise andmemudeli realisatsiooni. See keel sisaldab alamkeeli andmete töötlemiseks, andmebaasiobjektide haldamiseks, transaktsioonide haldamiseks, õiguste haldamiseks, protseduuride loomiseks.
SQL-andmebaasi-süsteem	***SQL database management system*** Andmebaasisüsteem, kus saab kasutada SQL-andmebaasikeelt.
SQL-andmebaas	***SQL database*** SQL-andmebaasisüsteemis loodud loodud andmebaas.
Täitmisplaan	***Query plan*** Deklaratiivse andmebaasikeele lause tulemuste saavutamiseks vajalik protseduur, mis kasutab süsteemi sisemise taseme operatsioone ning realiseerib deklaratiivse lausega soovitud tulemuseni jõudmiseks mõeldud algoritmi. Andmebaasisüsteemis, kus kontseptuaalne ja sisemine tase on üksteisest loogiliselt eraldatud ning andmete kasutajad töötavad andmetega kontseptuaalsel tasemel, koostab selle plaani andmebaasisüsteem. Andmebaasisüsteemi eesmärk pole mitte ainult valida esimene võimalik plaan, vaid üritada leida tulenevalt süsteemile optimeerimiseks seotud eesmärkidest, võimalikult hea plaan. Kuna kuitahes hea plaan võib olude sunnil halvaks osutuda, siis võib andmebaasisüsteemidel olla ka võime seda plaani lause täitmise käigus jooksvalt muuta.
WS	***Writeable Store*** *C-Store* tarkvarakomponent, mis pakub kiiret andmete sisestust.

Antud mõistete kirjeldamisel on kasutatud IT E-teadmiku Vallaste abi (E-Teadmik, 2015).

Jooniste nimekiri

Tabelite nimekiri

Sisukord

1. Sissejuhatus

Tänapäevases arvutikeskses maailmas tekib iga hetk tohutul hulgal informatsiooni. IBM'i hinnangul tekitati aastal 2014 iga päev 2,5 miljardi gigabaiti (2.5 ekasbaidi) andmeid ning see andmehulk on kasvanud iga aasta. (Jewell, et al., 2014) Suurt osa mängib siin ka sotsiaalmeedia. Näiteks Facebook'is tehakse iga päev 4,75 miljardit postitust ja 4,5 miljardit *like*'i ning Twitter'i kasutajad teevad päevas 400 miljonit säutsu. Aina rohkem kasvab ka nutitelefonide mõju, sest mobiilsete seadmete poolt tekitatakse päevas 5 miljonit gigabaiti andmeid. (Zikopoulos, et al., 2015) Lisaks sotsiaalmeediale toodavad tohutus koguses andmeid ka teaduslikud eksperimendid. Näiteks LHC (*Large Hadron Collider*) osakeste põrguti CERN'is (*Conseil Européen pour la Recherche Nucléaire* e Euroopa Tuumauuringute Keskus) toodab 30 miljonit GB aastas (Computing, 2015). Veelgi rohkem hakkab andma andmeid plaanitav rahvusvaheline raadioteleskoop SKA (*Square Kilometre Array*), 70 miljonit GB aastas. (Norris, 2010) Lisaks võib järgnevatel aastatel informatsiooni hulga kasvu üha rohkem mõjutama hakata ka IOT (*Internet of things*) e asjade internet. Selliste andmehulkade, mida lisaks iseloomustab katkematu voog, erinevate andmeformaatide paljusus, mahu kasv ajas ning peaaegu reaalajas töötlemise vajadus, kohta käib lööksõna *Big Data* e suurandmed. Suurandmed kujutab endas nii uusi võimalusi kui ka uusi väljakutseid. Väljakutseteks on leida tehnilised lahendused, kuidas nii suurt hulka andmeid salvestada ja piisavalt kiirelt töödelda. Suurandmete kiire analüüs aga võimaldab meil neid protsesse, mille kohta analüüsitavad andmed käivad, operatiivselt ja võib-olla isegi reaalajas efektiivsemaks muuta. (Jewell, et al., 2014)

1.1 Taust ja probleem

Küllaltki suur osa digitaalsetest andmetest hoitakse tänapäeval SQL-andmebaasides. Enamik nendest on loodud kasutades juba pikalt turul olevaid süsteeme. Neid on hakatud nimetama ka „traditsioonilisteks" või „vanadeks" SQL süsteemideks (*OldSQL systems*). Kahjuks aga ei ole need suurandmete töötlemiseks kõige efektiivsemad. Selleks, et efektiivsust (töökiirus, paralleeltöö võimalused, andmete salvestamiseks kulutatav ruum) märkimisväärselt parandada, tuleb süsteemide sisemistes tööpõhimõtetes teha muudatusi. Selle tööga tahan muuhulgas näidata, et taolise eesmärgi saavutamiseks pole sugugi tingimata vaja muuta

andmete esitamise ja töötlemise loogilist mudelit (relatsiooniline/SQL mudel) või loobuda deklaratiivse andmebaasikeele eelistest nagu paljud NoSQL süsteemide pooldajad kipuvad väitma. Leides uusi ja paremaid viise andmete andmebaasi sisemisel tasemel esitamiseks ja töötlemiseks saab loodetavasti parandada süsteemi efektiivsust ning samas säilitada võimalus töötada andmetega piisavalt kõrgel abstraktsioonitasemel. Sisemise taseme all mõeldakse sisemist taset ANSI-SPARC kolmekihilise andmebaasisüsteemide arhitektuuri seisukohalt lähtudes (SPARC/DBMS Study Group, 1975).

Kõikidele andmebaasisüsteemidele on omane andmete kirjutamine ja lugemine kettalt (ka täielikult muutmälu põhiste andmebaasisüsteemide puhul peavad andmed lõpuks kettale püsisalvestusse jõudma). Kettalt andmete lugemine või sinna kirjutamine on aga aeglane võrreldes muutmäluga, seega on see hea koht, kus muudatused saaksid andmebaasisüsteemide töökiirust märkimisväärselt mõjutada.

Isegi kui töödeldavad andmed ei kvalifitseeru suurandmeteks on ikkagi väga oluline, et infosüsteemide kasutajad saaksid enda küsimustele võimalikult kiiresti vastused. Teisisõnu on vaja leida parem viis kuidas salvestada andmeid kettale, et nende lugemine oleks kiirem. Samas muutes päringuid kiiremaks ei tohiks selle arvelt teised, samas andmebaasis toimuvad, operatsioonid aeglasemaks muutuda. Operatiivandmete andmebaasides, mida kasutavad onlain tehingutöötluse süsteemid (OLTP), on sellisteks operatsioonideks näiteks üksikute ridade lisamine, muutmine ja kustutamine.

Traditsioonilised SQL-andmebaasisüsteemid kasutavad reapõhist salvestamist. See tähendab, et andmebaasisüsteem salvestab sisemiselt andmefailides ühele leheküljele ühe tabeli null või rohkem rida. Erinevate tabelite read salvestatakse erinevatele lehekülgedele. Alternatiivne võimalus oleks kasutada veerupõhist andmete salvestamist, kus andmebaasisüsteem salvestab sisemiselt ühele leheküljele andmeid ainult ühest tabeli veerust. Iga veeru andmete salvestamiseks kulub null või rohkem lehekülge.

Päringu kiiruste parandamine on oluline, sest suurandmete peal tehtud analüüsid võimaldavad meil ette ennustada nii tarbija käitumist kui ka võimalikke tuleviku sündmusi. Suurte ettevõtete puhul võib iga väiksemgi lisateave oma klientide ja nende tarbimisharjumuste kohta tähendada suurt lisatulu. Mõttekoja *demosEUROPA* hinnangul mõjutavad suurandmed potentsiaalselt enam kui poolt Euroopa Liidu majandusest. Lisaks andmetepõhine otsustamine suurendaks ettevõtte efektiivsust konkurentidega võrreldes kuni 6%. Ennustatakse, et aastaks

2020 toob suurandmete kasutamine, täpsemalt nende põhjal tehtud otsused, Euroopa Liidu majandusele ligi 200 miljardi euro väärtuses kasu. (Mis ja milleks on big data?, 2015)

1.2 Ülesande püstitus

SQL-andmebaasisüsteemides saab realiseerida veerupõhise andmete salvestamise. Sellisel juhul töötab kasutaja tabelitega, kuid andmebaasi sisemisel tasemel salvestatakse andmeid veergude kaupa. Viimastel aastatel on loodud ka teistsuguseid süsteeme, kus andmeid organiseeritakse kasutajatele nähtavalt näiteks veergude perekondadena (*column family*). Selliste NoSQL süsteemide uurimine pole käesoleva töö eesmärgiks, kuid nende võrdlemine veerupõhist salvestust pakkuvate SQL-andmebaasisüsteemidega võiks olla eraldi lõputöö teemaks või antud töö edasiarenduseks.

Käesolevas töö eesmärk on uurida, kuidas mõjutab veerupõhine ja reapõhine andmete salvestamine kahes SQL-andmebaasisüsteemis loodud andmebaasides toimuvate andmekäitluse operatsioonide jõudlust ning nende andmebaaside andmemahte. Andmete salvestamise viisi määramine on andmebaasi füüsilise disaini ülesanne ja selle töö tulemus peaks aitama selle protsessi käigus paremaid valikuid teha. Töö alguses püstitatakse kirjanduse põhjal üldised hüpoteesid rea- ja veerupõhise andmete salvestamise kohta, mille kehtivusele antakse töö lõpuks konkreetsete süsteemide ja praktiliste katsete põhjal hinnang. Selle eesmärgi saavutamiseks on kõigepealt vaja uurida teoreetiliselt, milline on veerupõhine andmete salvestamise meetod ning millised on selle meetodi teoreetilised eelised ja puudused.

1.3 Metoodika

Veerupõhise andmete salvestamise meetodiga tutvutakse eelkõige teadusartiklite baasil. Kirjanduse kaasabil on võimalik aru saada veerupõhise andmete salvestamise meetodi teoreetilistest eelistest ning selle puudustest. Kirjanduse põhjal lõin järgnevad hüpoteesid, mida hakkan töö praktilises osas kontrollima. Need hüpoteesid võrdlevad omavahel veeru- ja reapõhise salvestusega andmebaasisüsteeme.

- Veerupõhise andmete salvestamisega andmebaasisüsteemides töötavad kiiremini päringud, mis töötlevad palju ridu aga samas kasutavad ja tagastavad vähe veerge. Antud töös pean päringu poolt töödeldavate ridade arvu suureks, kui päringu

predikaadile (`WHERE` klauslis olevale avaldisele) vastab 10% ja rohkem tabeli ridu. (Harizopoulos, et al., 2006)

- Veerupõhise andmete salvestamisega andmebaasisüsteemides päringu poolt tagastatav või kasutatav veergude hulk mõjutab nähtavalt päringu kiirust. Mida rohkem veerge on vaja, seda aeglasem peaks päring olema. (Harizopoulos, et al., 2006)

- Veerupõhise andmete salvestamisega andmebaasisüsteemidel ei tohiks olla erilist eelist, kui päringud töötlevad vähe ridu. Antud töös pean päringu poolt töödeldavate ridade arvu väikeseks, kui päringu predikaadile vastab 1% või vähem tabeli ridu. (Larson, et al., 2011)

- Reapõhise andmete salvestamisega andmebaasisüsteemid peaksid olema oluliselt kiiremad, kui üles on vaja leida ainult üks rida (andmed ühe olemi kohta), eeldades et `WHERE` klauslis kasutatav veerg on indekseeritud ning selles olevad väärtused on unikaalsed. (Larson, et al., 2011)

- Reapõhise andmete salvestamisega andmebaasisüsteemides toimub uute ridade lisamine kiiremini. (Stonebraker, et al., 2005)

- Reapõhise andmete salvestamisega andmebaasisüsteemides toimub ridade kustutamine kiiremini. (Stonebraker, et al., 2005)

- Reapõhise andmete salvestamisega andmebaasisüsteemides toimub ridade muutmine kiiremini. (Stonebraker, et al., 2005)

- Veerupõhise salvestamisega andmebaasisüsteemides loodud andmebaasid peaksid andmemahult olema oluliselt väiksemad. (Larson, et al., 2011)

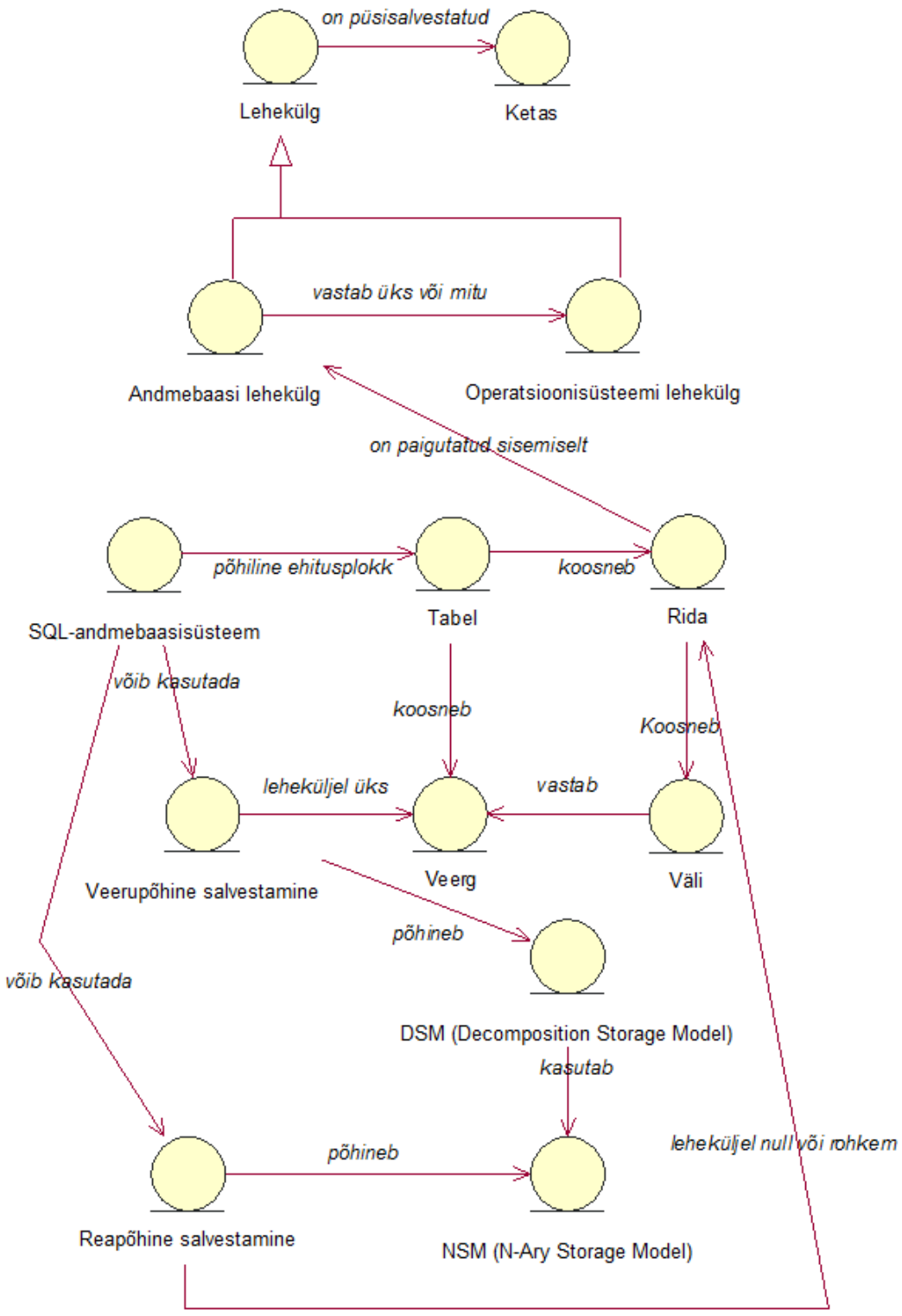
Hüpoteeside kinnitamiseks või ümberlükkamiseks projekteerin onlain tehingutöötluse süsteemi jaoks mõeldud operatiivandmete andmebaasi ning realiseerin selle kahes erinevas SQL-andmebaasisüsteemis. Andmebaasid täidan võimalikult suure hulga andmetega, et erinevused paremini välja tuleksid. Nendes andmebaasides viiakse läbi mitmesuguseid teste. Testideks on erinevate päringute ja andmemuudatuste käivitamine ja nende töötlemise aja mõõtmine. Nende päringute hulka kuuluvad nii tüüpiline tehingutöötluse päring (üksiku olemi andmete otsimine) kui ka koondandmete päringud, mida võib kohata nii tehingutöötluse kui

ka analüüsisüsteemides. Testitavad andmemuudatused on pigem iseloomulikud tehingutöötluse süsteemidele, sest muudetakse ühte rida korraga.

1.4 Ülevaade tööst

Töö on jaotatud nelja suurde ossa. Esimeses osas tutvutakse kirjandusega. Teemat puudutavate mõistete ja nende seoste selgemaks esiletoomiseks on kasutatud UML klassidiagrammidel põhinevaid mõistekaarte. Teises osas tutvutakse erinevate veerupõhiste andmebaasisüsteemidega ja uuritakse, kuidas on nendes realiseeritud veerupõhine andmete salvestamine. Kolmandas osas luuakse andmebaasid kahes erinevas SQL-andmebaasisüsteemis ja viiakse nendes läbi erinevaid teste. Neljandas osas analüüsitakse eksperimendi käigus leitud tulemusi. Töö lõppeb järelduste ja kokkuvõttega.

2. Andmete salvestamine SQL-andmebaasisüsteemides



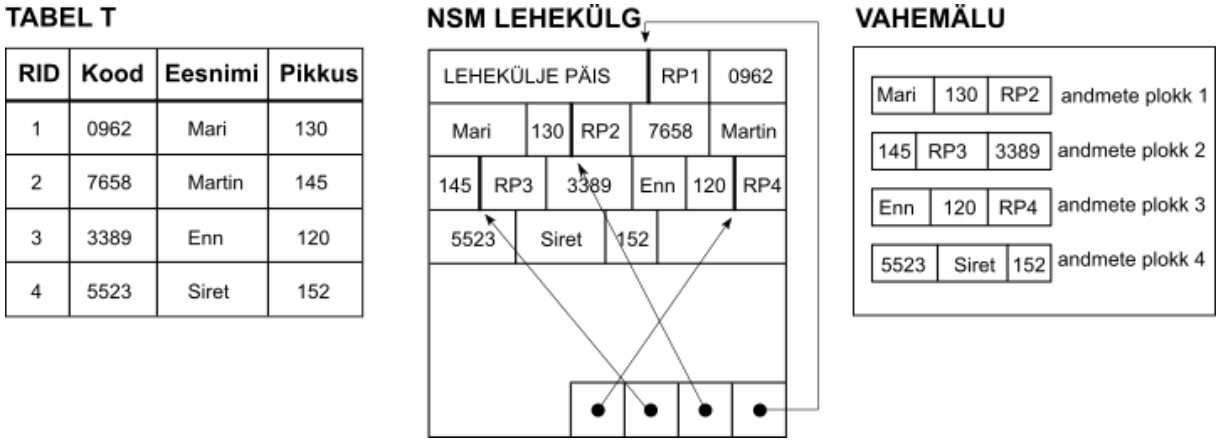
Joonis 1. Andmete salvestamise meetodite mõistekaart

ANSI-SPARC andmebaaside ja andmebaasisüsteemide kolmekihilise arhitektuuri kohaselt kuuluvad andmete salvestamisega seotud küsimused andmebaasi sisemisele tasemele. See tähendab, et muutes andmebaasi sisemisel tasemel andmete salvestamise viisi, ei muutu loogilisel e kontseptuaalsel tasemel olevate tabelite kasutaja jaoks midagi. See on väga tähtis, sest nii välditakse andmebaasi kasutavate rakenduste ümberkirjutamist, mis omakorda lihtsustab oluliselt selliste muudatuste kasutuselevõttu. SQL-andmebaasisüsteemid on üldjuhul kasutanud reapõhist salvestamist, kuid nüüd on esile kerkimas ka veerupõhine salvestamine (vt joonis 1).

2.1 Reapõhine andmete salvestamine SQL-andmebaasisüsteemides

Enamik tänapäeva tuntumaid SQL-andmebaasisüsteeme kasutab reapõhist andmete salvestamist. See tähendab, et tabeli null või rohkem rida salvestatakse sisemiselt ühele leheküljele. Lehekülg on baitide jada, mida operatsioonisüsteem käsitleb ühe tervikuna. Operatsioonisüsteem loeb ja salvestab andmeid lehekülgede, mitte üksikute baitide haaval. Windows NTFS failisüsteemis on tavaliselt ühe lehekülje suuruseks 4KB. Lehekülgede kaupa andmete haldamine toob kaasa mõningase kõvaketta ruumi ebaefektiivse kasutamise, sest ka ühte baiti sisaldava faili suurus on kõvakettal 4KB. Andmebaasisüsteemide lehekülgede suurused aga ei pruugi vastata üks-ühele operatsioonisüsteemi omaga. Näiteks Microsoft SQL Serveris on ühe lehekülje suuruseks 8KB (Harizopoulos, et al., 2006).

Üks tuntumaid andmebaasisüsteemide salvestusmudeleid on *Slotted pages* või NSM (*N-Ary Storage Model*). NSM'i iga lehekülg koosneb lehekülje päisest (*page header*), järjestikustest lehekülje elementidest ja päise järel või lehekülje lõpus asetsevatest pesadest (*slot*). Lehekülje päises on kirjas üldine informatsioon lehe ja temas asetsevate elementide kohta, iga elemendi ees olevas päises RP (*header*) on kirjas näiteks nihked (*offset*) muutuvate suuruste väärtuste jaoks ja muu vajalik informatsioon. Pesades on kirjas viit iga elemendi algusesse. See on vajalik, sest elemendid võivad olla erineva suurusega (vt joonis 2). Viida asukoht määrab ära mitmenda elemendiga on tegemist. (Ailamaki, et al., 2002)



Joonis 2. Tabel T, NSM lehekülg ja vahemälu
(Ailamaki, et al., 2002)

Lehekülje elemendiks võib olla rida (kui leheküljel salvestatakse tabeli andmeid) või indeksi sissekanne (kui leheküljel salvestatakse indeksi andmeid). Selline salvestusmudel võimaldab muuta elemendi paigutust lehekülje sees ilma, et muudetakse elemendi füüsilist aadressi, mida kasutatakse väljapool lehekülge elemendile viitamiseks.

Sellisel mudelil esinevad aga mõningad puudujäägid. Võtame vaatluse alla päringu, mis leiab üles kõikide isikute eesnimed tabelist T, kelle pikkus on väiksem kui 140 cm.

```
SELECT eesnimi FROM T WHERE pikkus < 140;
```

Kõigepealt otsib protsessor vajalikke andmeid vahemälu hierarhiast (L1, L2, L3), kui neid sealt ei leita (*cache miss*) siis viiakse need sinna muutmälust (*RAM*), kui neid ei ole veel ka muutmälus, siis tuleb need kettalt muutmällu lugeda. Protsessori vahemälust lugemine on kordades kiirem kui arvuti muutmälust ja muutmälust jällegi oluliselt kiirem kui kettalt. Tänapäeval on serveritesse võimalik panna piisavalt muutmälu, et kogu andmebaas või suurem osa sellest sinna ära mahutada, vahemälu on seevastu aga väga väike ja seda peaks kasutama võimalikult optimaalselt. Vahemälu koosneb ridadest (*cache entry* e *cache line*), mille suurus oleneb protsessorist, kuid jääb tavaliselt umbes 100 baidi juurde. Vahemälu enda suurus sõltub tasemest, olles esimese taseme puhul paarkümmend kilobaiti, teise puhul paarsada kilobaiti ja kolmanda puhul paar megabaiti. Antud päringu ja salvestusmudeli puhul ei kasutata vahemälu efektiivselt, sest vahemällu viiakse selle päringu kontekstis üleliigseid andmeid. (vt joonis 2, kolmas pilt). Kui päringu predikaat (`WHERE` klauslis olev

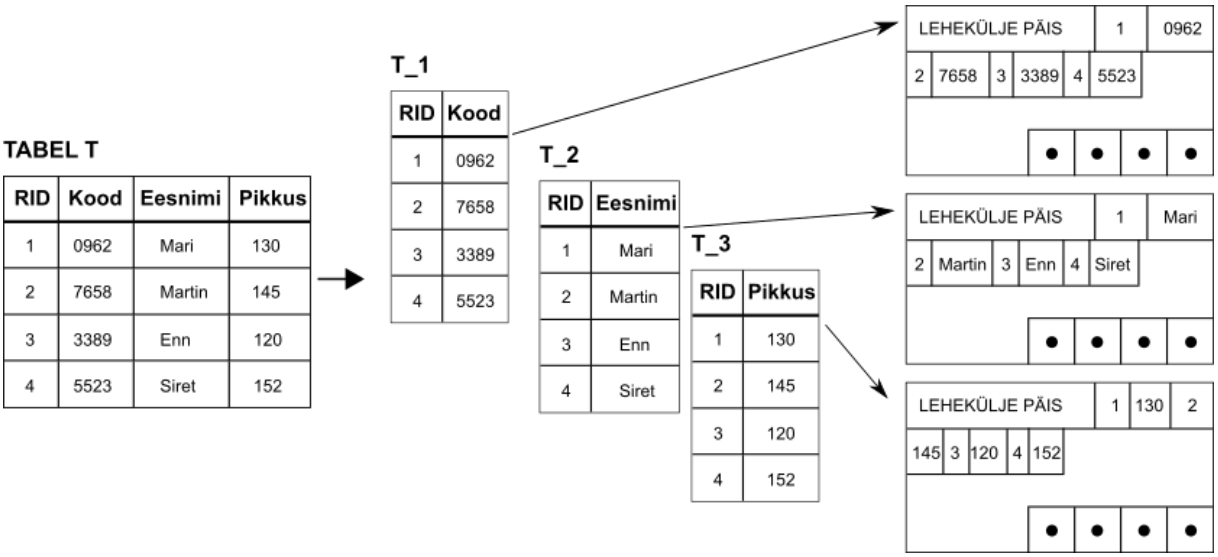
otsingutingimus) sisaldaks kõiki tabeli veerge, siis ei oleks see enam probleemiks, sest päringu täitmiseks on vaja lugeda andmeid kõigist veergudest. (Ailamaki, et al., 2002)

NSM salvestusmudeli eeliseks on aga hea andmete salvestamise kiirus, kuna uue rea lisamine toimub ühe korraga, sest rea andmeid pole üldjuhul vaja kirjutada mitmele leheküljele. Lisaks jäetakse lehekülgedele ka tavaliselt vaba ruumi, et seal olevad read saaksid kasvada (ridade migreerimine uuele leheküljele on kulukas). Sellepärast nimetatakse ka NSM või selle sarnast salvestusmudelit kasutavaid andmebaasisüsteeme kirjutamisele orienteerituks (*write-optimized*).

2.2 Veerupõhine andmete salvestamine SQL-andmebaasisüsteemides

Arvutiriistvara arengus paistab välja tendents, et protsessori kiirus kasvab kiiremini kui muutmälu või ketta oma. Sellest järeldub veel üks võimalus, kuidas andmebaasisüsteemides operatsioonide töökiirust tõsta. Vahetades ketta lugemise ja kirjutamise operatsioone protsessori operatsioonide vastu peaks andmebaasisüsteemi üldine jõudlus paranema. (Halverson, et al., 2006). Veerupõhise salvestamisega andmebaasisüsteemid tavaliselt seda seaduspära paremate päringu täitmiskiiruste saavutamiseks ära kasutavadki, tavaliselt andmete kodeerimise ja pakkimise läbi.

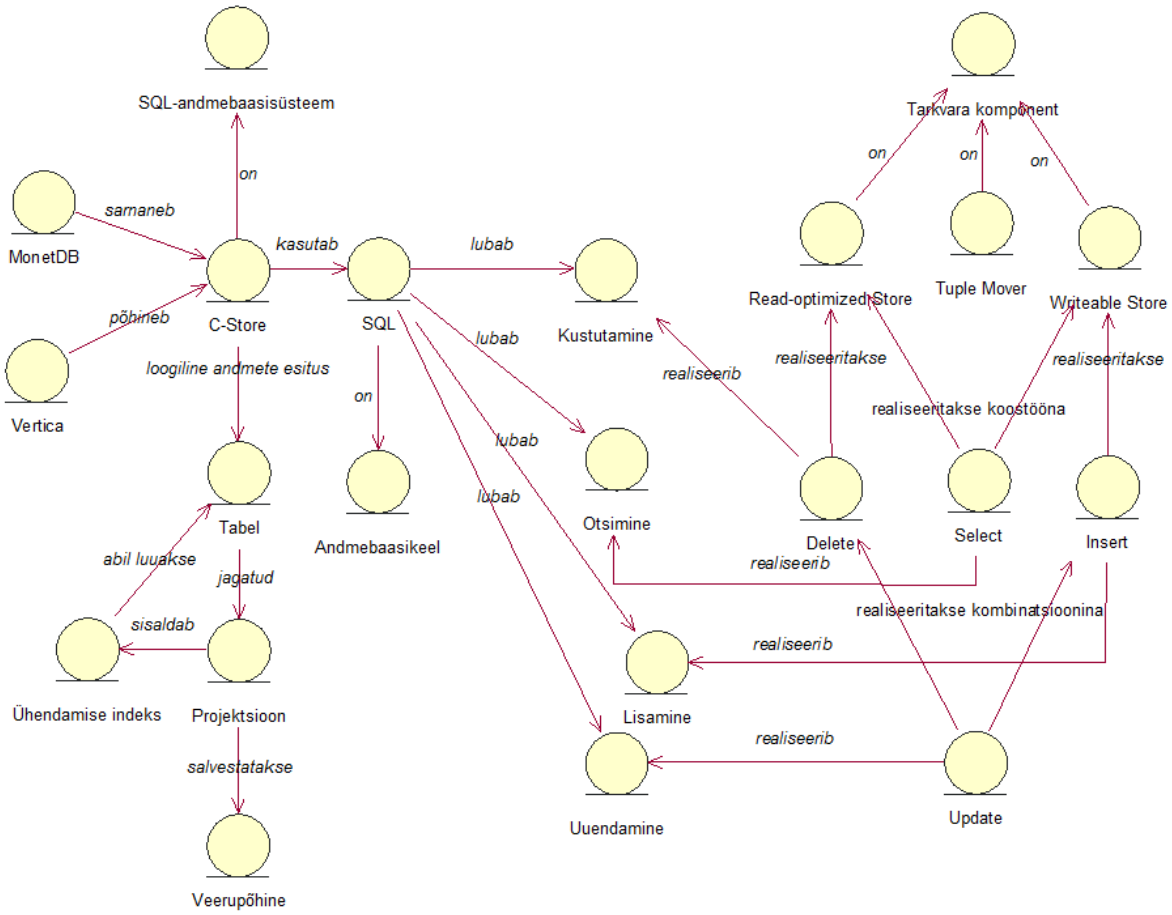
Üks esimesi veerupõhise andmete salvestusega süsteemidele loodud salvestusmudeleid oli DSM (*Decomposition Storage Model*). Mudeli eesmärgiks oli vähendada andmete hulka, mida oli vaja lugeda kettalt ning samuti suurendada protsessori vahemälu kasutamise efektiivsust. (Harizopoulos, et al., 2006) DSM'i puhul jagatakse (killustatakse) tabel vertikaalselt veergude kaupa tükkideks (dekomponeeritakse kasutades projektsiooni operaatorit) selliselt, et alamtabelitesse jäävad rea füüsilised identifikaatorid e aadressid ja veergudele vastavad väärtused. Need alamtabelid aga salvestatakse NSM meetodi abil (vt joonis 3). (Ailamaki, et al., 2002) Kasutaja näeb endiselt terviklikku tabelit, sest tükeldamine toimub automaatselt andmebaasi sisemisel tasemel.



Joonis 3. Tabeli tükeldamine DSM mudeli korral
(Ailamaki, et al., 2002)

DSM salvestusmudeli negatiivseteks külgedeks on suurenenud salvestusruum, kuna iga veeru väärtuse juurde salvestatakse rea füüsiline identifikaator (RID). Samuti on negatiivseks küljeks ka teatud juhtudel aeglane päringu operatsioonide täitmise aeg, kui päringu vastuses on vaja esitada andmeid paljudest veergudest, sest see tähendab mitme erineva lehekülje lugemist. (Ailamaki, et al., 2002) Lisaks muutuvad aeglasemaks ka uute ridade lisamine, sest rea väljades olevad väärtused on vaja kirjutada erinevatesse lehekülgedesse. Päringud, mis aga vajavad vähe veerge, peaksid olema aga oluliselt kiiremad kui reapõhise salvestusega andmebaasisüsteemides. Selletõttu kutsutakse DSM või sarnast salvestusmudelit kasutavaid andmebaasisüsteeme lugemisele orienteerituks (*read-optimized*).

3. Veerupõhine andmebaasisüsteem *C-Store* näitel



Joonis 4. *C-Store* mõistekaart

Praeguseks hetkeks eksisteerib juba mitmeid SQL-andmebaasisüsteeme, mis kasutavad veerupõhist salvestamist. Mõned neist põhinevad või järgivad eksperimentaalse andmebaasisüsteemi *C-Store* (vt joonis 4) disaini. Selliste süsteemide näideteks on kommertslikest andmebaasisüsteemidest *Vertica* (Lamb, et al., 2012) ja vabavaralistest MonetDB. *C-Store* töötati välja mitme ülikooli koostööna ning seda on kirjeldatud juba 2005 aastal ilmunud teadustöös. (Stonebraker, et al., 2005)

C-Store on väljastpoolt kui tavaline SQL-andmebaasisüsteem. See tähendab, et andmete loogiline esitus toimub tabelitena ja andmebaasikeeleks on SQL. Sisemiselt on aga *C-Store* teistsuguse ehitusega kui traditsioonilised SQL-andmebaasisüsteemid. Esiteks andmed

salvestatakse veergude kaupa e tervikliku tabeli asemel salvestatakse veergude projektsioonid. Teiseks toimub andmete pakkimine (*densepack*) ja kodeerimine.

C-Store salvestab tabelid projektsioonidena ja iga veerg projektsioonis on salvestatud veerupõhiselt. Projektsiooniks nimetatakse veergude gruppi, mis on sorteeritud ühe või mitme projektsioonis asuva veeru järgi. Projektsioon on ankurdatud ühe tabeli külge, kuid võib sisaldada veerge ka välisvõtme kaudu seotud tabelitest. Projektsioonis on sama palju ridu kui selle ankurtabelis, st projektsioonide korral ei toimu korduvate ridade eemaldamist. Sorteerimiseks võib kasutada ka veergu, mis asub välisvõtmega viidatud tabelis. Üks ja sama veerg võib esineda mitmes projektsioonis ning iga veerg peab esinema vähemalt ühes projektsioonis. See tähendab, et andmebaasi sisemisel tasemel esineb andmete liiasust. Kuna *C-Store* kasutab sisemiselt andmete pakkimist, siis ei tohiks andmete liiasus põhjustada suurt andmebaasi andmemahu kasvu. Samas annavad need dubleeritud veerud, kui nad on projektsioonides erinevalt sorteeritud, täiendavaid võimalusi päringute optimeerimiseks. Dubleeritud projektsioone saab näiteks jagada erinevate sõlmede/serverite (*node*) vahel, mis annab *C-Store*'le kiire andmete edastuse ja kõrge tõrketaluvuse, seda juhul kui igas sõlmes on piisavalt projektsioone, et terve tabel kokku panna. Näiteks töötaja ja osakonna tabeli järgi (vt joonis 5) saaks moodustada järgnevad projektsioonid (peale püstkriipsu on ära toodud veerg, mille järgi antud projektsioon on sorteeritud). Need on ainult ühed võimalikud projektsioonide näited.

- TÖÖTAJA_1 (Eesnimi, Pikkus | Pikkus)
- TÖÖTAJA_2 (Osakond, Pikkus, Osakond.Korrus | Osakond.Korrus)
- TÖÖTAJA_3 (Eesnimi, Palk | Palk)
- OSAKOND_1 (Nimi, Korrus | Korrus)

TÖÖTAJA

Eesnimi	Pikkus	Osakond	Palk
Mari	130	Matemaatika	2200
Martin	145	Füüsika	2000
Enn	120	Geneetika	2500
Siret	152	Informaatika	3000

OSAKOND

Nimetus	Korrus
Matemaatika	3
Füüsika	2
Geneetika	1
Informaatika	4

Joonis 5. Töötaja ja osakonna tabelid

(Stonebraker, et al., 2005)

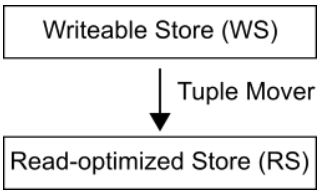
Lisaks on projektsioonid ka horisontaalselt killustatud segmentideks. Igal segmendil on oma identifikaator SID. Killustamine põhineb sorteerimise aluseks olnud veeru väärtuse vahemikel. Selleks, et projektsioonidest algset rida kokku panna, haldab *C-Store* ühendamise indekseid (*Join Indeces*). Antud indeks sisaldab SID'd ja SK'd (*Storage key*). SK'd ei ole füüsiliselt salvestatud, vaid tuletatud rea järjekorranumbrist projektsioonis. Väärtused, mis on samas segmendis, vastavad erinevatele veergudele, kuid millel on sama SK, moodustavad loogilise rea. Kõik ühendamise indeksid on sama tabeli külge ankurdatud projektsioonide vahel ja seega kirjeldavad need üks-ühele vastavusi. Eelneva näite puhul oleks näiteks vaja luua ühendamise indeks, mis kirjeldab projektsiooni TÖÖTAJA_2 (T_2) suhet projektsiooni TÖÖTAJA_3 (T_3) (vt tabel 1). Lihtsuse mõttes eeldame, et projektsioonid ei ole horisontaalselt killustatud st SID = 1 kõikide ridade puhul.

Tabel 1. Projektsioonide ühendamise indeksit kujutav tabel

SID (projektsioonis T_2)	**SK (projektsiooni T_3 vastavas segmendis)**
1	3
1	1
1	2
1	4

Indeksit (tabelit) tuleks lugeda järgmiselt. Projektsiooni T_2 esimene segmendi esimene rida vastab projektsiooni T_3 esimese segmendi kolmandale reale, teine rida projektsiooni T_3 esimese segmendi esimesele reale jne. Niimoodi saab indeksite alusel erinevatest projektsioonidest terve tabeli *Tootaja* kokku panna. Kuigi iga päringu jaoks sobilik projektsioon oleks hea, on nende loomine ja haldamine siiski kulukas ja seega tuleks neid luua võimalikult vähe.

Tabeli jaotamine projektsioonideks toob kaasa aga andmete lisamise ja uuendamise keerukuse kasvu. *C-Store* lahendab selle probleemi, kasutades mitut eraldiseisvat tarkvarakomponenti. Andmete uuendamise ja lisamisega tegeleb *Writeable Store* (WS). Andmete lugemisega tegeleb *Read-optimized Store* (RS). Andmete ülekandmisega WS'st RS'i tegeleb *Tuple Mover* (TM) (vt joonis 6). TM liigutab andmeid hulgakaupa, sest see on optimaalsem ning teeb seda automaatselt, kasutaja sekkumiseta.



Joonis 6. *C-Store* tarkvarakomponendid
(Stonebraker, et al., 2005)

Mõlema komponendiga käib kaasas eraldi andmehoidla, kus WS on optimeeritud andmemuudatuste tegemiseks ning RS andmete lugemiseks.

Antud arhitektuur tagab kiire andmete sisestuse kui ka lugemise võimalikult väikeste kompromissidega. Kompromissideks on, et päringute korral on vaja andmeid lugeda nii WS'ist kui ka RS'ist. Uusi ridu lisatakse üksnes WS'i, aga samas kustutatud read märgitakse ära RS'is. Uuendused (`UPDATE`) on realiseeritud lisamise (`INSERT`) ja kustutamise (`DELETE`) operatsiooni kombinatsioonina. Päringud, mis ainult loevad andmeid, käivitatakse ajalises isolatsioonis (*snapshot isolation*). Seda tehakse, et vältida lukkude kasutamist, mis vähendaksid oluliselt paralleelsete päringute kiiruseid. Ajaline isolatsioon tähendab, et päringud täitmiseks teeb andmebaasisüsteem ajarännu ajas tagasi kõige lähemasse hetke T, millal andmebaasis ei ole kinnitamata transaktsioonide tehtud muudatusi (st süsteem loeb garanteeritult terviklikke andmeid). Päringud, mis muudavad andmeid, toimivad range kahefaasilise lukustamise protokolli alusel ning muudatuste eelse seisu taastamise võimalus tagatakse logi abil (*write-ahead logging*).

C-Store üheks iseärasuseks on andmete kodeerimine. *C-Store* kasutab nelja erinevat veeru kodeerimise viisi. Kodeerimise viis oleneb sellest, millise veeru järgi on veeru enda projektsioon sorteeritud ning kui homogeensed on selle veeru väärtused. Alati siiski kodeerimist ei kasutata, näiteks kui veeru väärtused on liiga heterogeensed.

Esialgsed *C-Store* testid näitasid, et vaatamata andmete dubleerimisele kasutas *C-Store* kuni 40% vähem kettaruumi võrrelduna reapõhise salvestamisega andmebaasisüsteemi vastu. Keskmiselt oli *C-Store* lugemis päringute täitmisel 164 korda kiirem kui võrreldav reapõhist salvestamist kasutav populaarne (teadustöös nime ei täpsustatud) andmebaasisüsteem. Antud tulemus saavutati olukorras, kus andmebaasidele oli rakendatud ühine andmemahu piirang.

Lubades reapõhise salvestamisega andmebaasisüsteemis luua päringute täitmise abistamiseks materialiseeritud vaateid e hetktõmmiseid, siis päringute kiiruse vahe kahaneb keskmiselt 6,4 kordseks, aga siiski *C-Store* kasuks. Materialiseeritud vaadetega oli aga reapõhise andmebaasi andmemaht kuus korda suurem. (Stonebraker, et al., 2005) Materialiseeritud vaate e hetktõmmise korral on päringu tulemus eelnevalt välja arvutatud ning eraldi salvestatud. Materialiseeritud vaateid on vaja värskendada, et tagada nende kooskõla aluseks olevate tabelitega, kus andmed võivad muutuda.

C-Store tulemused on küll paljulubavad, aga siiski on tegemist kõigest kontseptsiooni tõestamiseks loodud prototüübiga, mitte kommertslikuks kasutamiseks mõeldud andmebaasisüsteemiga. Antud töö eksperimentaalse osa üheks eesmärgiks on samuti välja uurida, kas ka praegused veerupõhised andmebaasisüsteemid sellist jõudluse kasvu pakuvad.

4. Eksperimendis osalevad andmebaasisüsteemid

Töösse valitud andmebaasisüsteemide valiku kriteeriumiteks oli nende kättesaadavus ja populaarsus. Populaarsuse mõõtmiseks kasutati andmebaasisüsteemide hinnanguid kajastavat veebilehte *DB-Engines* (DB-Engines Ranking, 21.03.2015). *DB-Engines* kasutab andmebaasisüsteemi hinnangu loomiseks erinevaid mõõdikuid. Näiteks vaadatakse, kui palju on antud andmebaasisüsteemi mainitud erinevatel veebilehtedel (Google ja Bing), milline on otsingute sagedus ja üldine huvi (Google Trends), palju mainitakse seda tehnilises arutelus (Stack Overflow ja DBA Stack Exchange) ning kui palju on antud süsteemi mainitud tööpakkumistes ja erinevates suhtluskeskkondades (LinkedIn ja Twitter). Populaarsus on tähtis, kuna see näitab kogukonna jätkuvat huvi andmebaasisüsteemi vastu, mis omakorda näitab, et tegemist ei saa olla väga halva tootega. Uute (andmebaasi)süsteemide vastu langeb huvi kiiresti, kui need ei täida loojate poolt antud lubadusi või ei ole veel piisavalt küpsed (sisaldavad mitmeid kasutamist blokeerivaid vigu) reaalseks kasutamiseks. Lisaks, kasutades eksperimendis just kõige populaarsemaid andmebaasisüsteeme, on potentsiaalne kasu antud töö tulemustest suurim.

Antud töö eesmärgiks ei ole erinevate veerupõhise salvestamisega andmebaasisüsteemide üldine võrdlemine, vaid selgitada konkreetsete süsteemide ja näitebaasi põhjal välja veerupõhise salvestamise mõju andmekäitluse operatsioonide töökiirusele ja andmemahtudele. Seega oleks hea valida just sellised andmebaasisüsteeme, kus mõlemad võimalused on olemas.

Andmebaasisüsteemid, kus on mõlemad olemas on näiteks *DB-Engines* nimistus (detsembris 2015) teisel kohal asuv MySQL. MySQL andmebaasisüsteem ise salvestab andmeid reapõhiselt, kuid on olemas andmebaasisüsteeme, mis põhinevad MySQL'il, kuid salvestavad andmeid veerupõhiselt. Näiteks *DB-Engines* nimistus (detsembris 2015) 103 kohal asuv *Infobright* ja (detsembris 2015) 137 kohal asuv *InfiniDB.* Nende puhul on andmebaasimootoriks endiselt MySQLi mootor, kuid selle mõned tarkvarakomponendid nagu näiteks salvestusmootor ja päringute optimiseerija on ümberkirjutatud. Veel üheks selliseks andmebaasisüsteemiks on *DB-Engines* nimistus (detsembris 2015) kolmandal kohal asub Microsoft SQL Server, mis vaikimisi salvestab andmeid reapõhiselt, kuid milles on võimalik luua ka indeksit (*columnstore index*), mis salvestab andmed veerupõhiselt.

Oma töö eesmärkide täitmiseks piisab ainult ühest neist ja sobilikum on Microsoft SQL Server. Microsoft SQL Serveri puhul ei ole tegemist kahe eraldi tootega nagu seda on näiteks MySQL ja *Infobright* ja läbi *DreamSpark* programmi oli mul võimalus hankida Microsoft SQL Server 2014 *Enterprise* versioon tasuta. Lisaks ka ettevõttes, kus ma töötan on mitmete infosüsteemide puhul kasutusel just Microsoft SQL Server andmebaasisüsteem. Seega leidub võimalus, et antud töö tulemusi on võimalik rakendada oma praeguse töökoha juures.

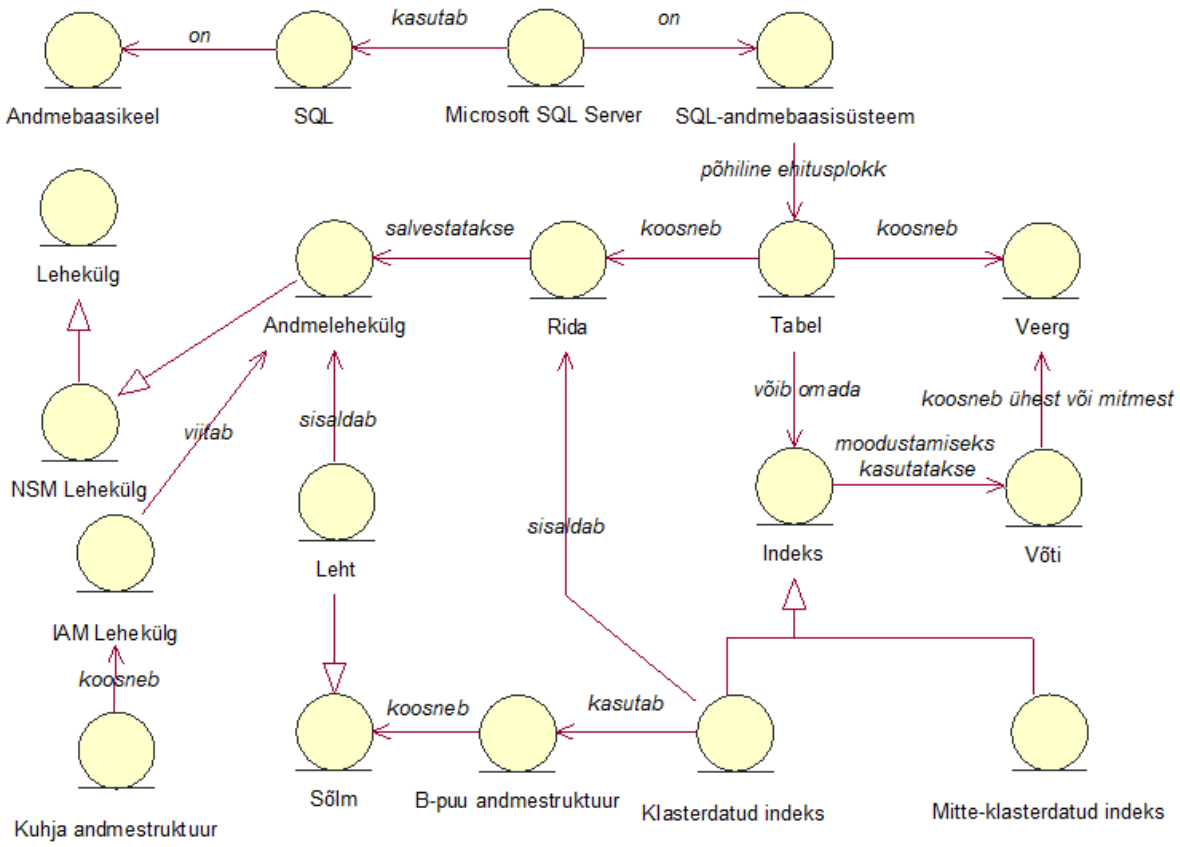
Kuna Microsoft SQL Serveri puhul ei ole tegemist puhta veerupõhise salvestamisega andmebaasisüsteemiga, st veerupõhine salvestamine lisati hiljem juurde, siis otsustasin testides kasutada ka ühte puhast veerupõhise salvestusega andmebaasisüsteemi. Teemaga tutvumiseks loetud teadusartiklites [(Abadi, et al., 2008), (Boncz, et al., 2005), (Harizopoulos, et al., 2006), (Ideros, et al., 2012)] oli mitmel neist mainitud andmebaasisüsteemi MonetDB (MonetDB, 2015). Kuna MonetDB on vabalt kättesaadav andmebaasisüsteem, siis osutus just see teiseks andmebaasisüsteemiks. MonetDB andmebaasisüsteem asub üldises *DB-Engines* nimistus (detsembris 2015) 105. kohal ning SQL-andmebaasisüsteemide nimistus (detsembris 2015) 52. kohal (kokku nimekirjas 109 SQL süsteemi).

4.1 Microsoft SQL Server

Töös kasutatud versioon: 12.0 (SQL Server 2014), Enterprise Edition, 64-bit

Andmebaasikeel: SQL

Andmete salvestamise meetod: reapõhine salvestamine ja veerupõhist salvestamist toetav indeksi tüüp.



Joonis 7. Microsoft SQL Server üldine mõistekaart

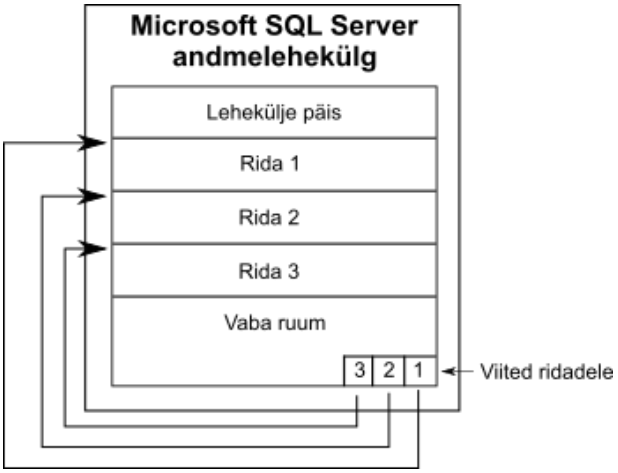
Microsoft SQL Server on SQL-andmebaasisüsteem (vt joonis 7), mis salvestab andmeid reapõhiselt, kuid milles on ka realiseeritud indeksi tüüp (*Column Store Index*, hiljem CSI), mis salvestab indekseeritud tabelid veerupõhiselt. Microsoft SQL Server tutvustas seda uut indeksi tüüpi versioonis 11.0 „Denali“ (SQL Server 2012). Microsoft SQL Server andmebaasisüsteemi vaadatakse antud töö eksperimendis kahe nurga alt. Esiteks kui nn traditsioonilist andmebaasisüsteemi, mis salvestab andmeid reapõhiselt. Teiseks kui andmebaasisüsteemi, milles on realiseeritud indeksi tüüp, et jäljendada veerupõhise salvestusega andmebaasisüsteeme. Selline valik annab väga hea võrdlusmomendi, erinevalt sellest kui reapõhise salvestusega andmebaasisüsteemi esindaks mõne teise ettevõtte või

arendajate grupi tarkvara, sest siis oleksid testide tulemused mõjutatud nii veerupõhise salvestamise meetodist kui ka süsteemide endi eripäradest.

Microsoft SQL Serveri 2014 on kommertstarkvaraga, see tähendab et toote kasutamise eest on vaja maksta, välja arvatud nende limiteeritud *Express* tarkvara liin. Töö jaoks on siiski saadud *Enterprise* tarkvara, seda läbi Microsofti *DreapSpark* programmi. Töö jaoks hankisin endale tarkvarad Microsoft SQL Server 2014 Enterprise Edition ja Microsoft SQL Server 2014 Management Studio.

4.1.1 Tabeli andmete salvestamine

Väikseim andmete salvestamise ühik Microsoft SQL Serveris on lehekülg. Ühe lehekülje suurus on 8KB. Microsoft SQL Serveris on kaheksa lehekülje tüüpi. Nende seast tabeli ridade salvestamiseks on mõeldud *Data* tüüpi lehekülg (andmelehekülg). Põhimõttelt on tegemist NSM lehekülgedega. Iga lehekülg (vt joonis 8) sisaldab 96 baidi suurust päist, kus on kirjas vajalik informatsioon antud lehekülje kohta nagu näiteks lehekülje number, tüüp, vaba ruum ja omaniku ID. Lehekülje päisele järgnevad kohe andmete read, kus hoitakse ridu moodustavaid väärtuseid. Seal on kirjas kõik väärtused peale väga suurte tekstiliste või binaarsete väärtuste. Lehekülje lõpus asetsevate viitade ja ridade vahele jääb võib jääda vaba ruum. Lehekülje lõpus asetsevad viited näitavad ridade algust selliselt, et tagantpoolt kõige esimene viit suunab selle lehekülje esimese rea algusesse ning tagantpoolt teine teise rea algusesse jne. (Understanding Pages and Extents, 2015) Antud lehekülje tüüp on kasutusel mõlemal juhul – nii siis kui tabel on salvestatud kuhja andmestruktuuri kui ka klasterdatud indeksisse.

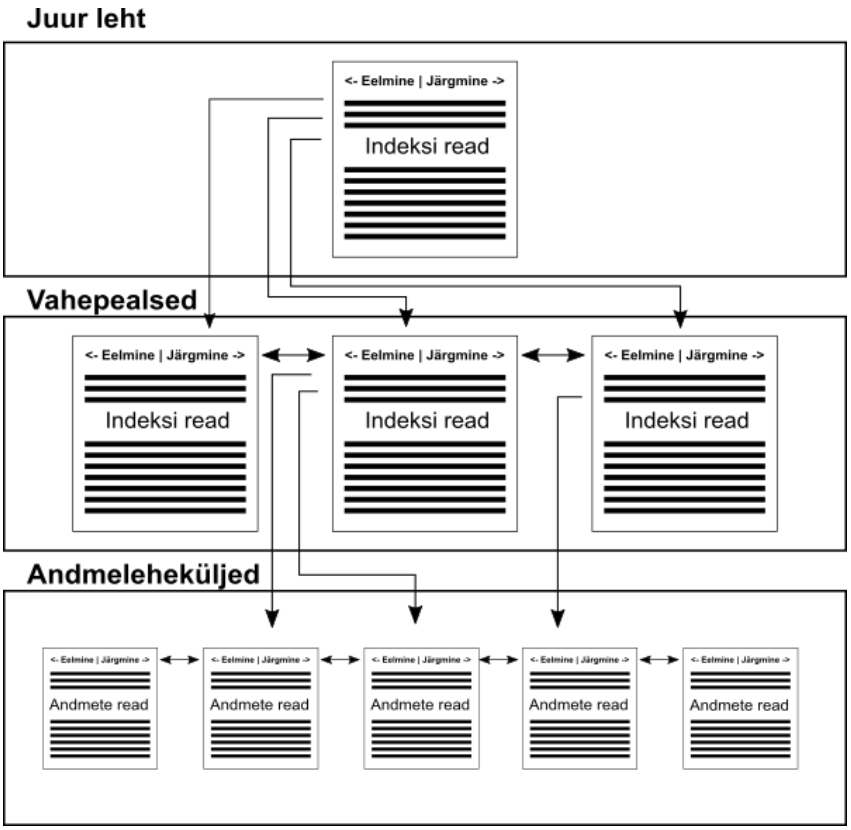


Joonis 8. Microsoft SQL Server'i andmelehekülg
(Understanding Pages and Extents, 2015)

4.1.2 Tavaline tabel ja klasterdatud indeksiga tabel

Kui tabelil pole ühtegi primaarvõtme indeksit ja tabelile pole loodud klasterdatud indeksit, siis salvestatakse tabel kuhja andmestruktuuri (*Heap structure*). Kuhja andmestruktuuri korral haldab Microsoft SQL Server sisemiselt tabeli kohta ühte või mitut IAM (*Index Allocation Map*) lehekülge, kus on omakorda viidad tabeli andmelehekülgedele. Andmeleheküljed ei ole IAM'is järjestatud. Samuti ei ole järjestatud ka read andmelehekülgedel ning andmelehekülgedel puuduvad viited järgmisele ja eelmisele leheküljele. See tekitab olukorra, kus sobilike andmete otsimiseks on vaja läbi vaadata kõik andmeleheküljed ja tagastatud read ei pruugi olla sisestamise järjekorras. (Heap Structures, 2014)

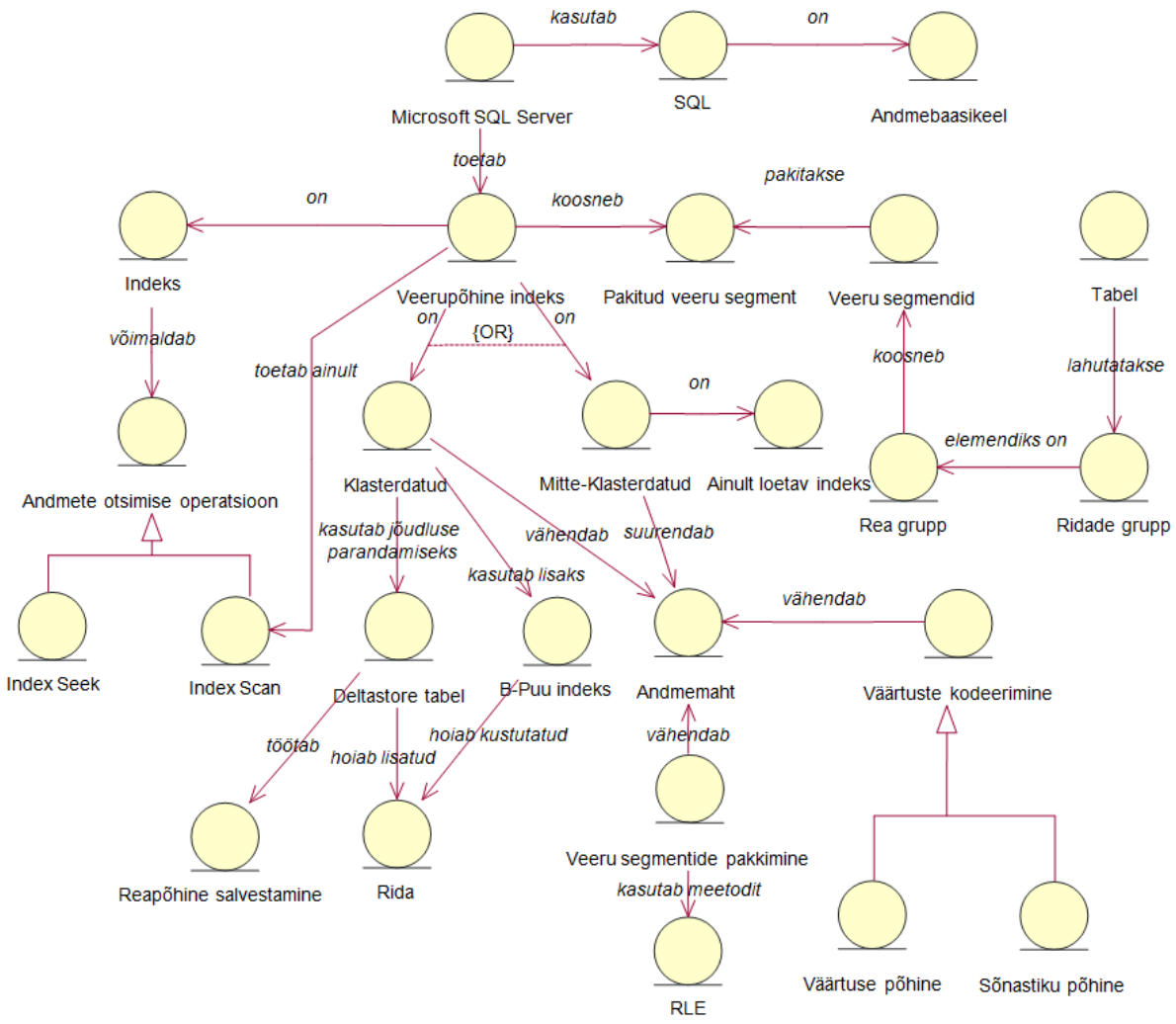
Kui tahta tabeli ridu salvestada sorteeritult on üheks võimaluseks luua tabelile klasterdatud indeks. Tabelil saab olla ainult üks klasterdatud indeks, sest tabeli read saavad füüsiliselt olla sorteeritud ainult ühes järjekorras. Klasterdatud indeks luuakse ühele veerule või veergude kombinatsioonile. See võib, kuid ei pea, olema primaarvõtme veerg. Klasterdatud indeksi veergu või veerge nimetatakse indeksi võtmeks. Klasterdatud indeks kujutab endast tavaliselt B-puu andmestruktuuri, kus vahepealsetes sõlmedes asetsevad indeksi leheküljed ning lõpp sõlmedes (lehtedes) asetsevad andmeleheküljed („Data" tüüpi leheküljed). Indeksi lehekülgedes on kirjas klasterdatud indeksi võtme väärtused ja viited eelmisele/järgmisele indeksi leheküljele (vt joonis 9). Nii võtmeväärtused indeksi lehekülgedel kui ka read andmelehekülgedel on sorteeritud vastavalt klasterdatud indeksi võtme väärtusele. (Clustered Index Structures, 2015)



Joonis 9. Klasterdatud indeksi ehitus Microsoft SQL Serveris
(Clustered Index Structures, 2015)

Kui tabelile on loodud klasterdatud indeks, siis ka päringu, mille täitmine nõuab andmebaasisüsteemilt terve tabeli läbivaatamist, toimub mööda indeksit ning selle käigus loetakse kõiki indeksipuu lehti. Microsoft SQL Serveris on indeksi täieliku läbivaatamise tingiva sisemise taseme otsimise protseduuri nimeks *index scan*. Juhul kui tabelil ei ole ühtegi indeksit, siis rakendatakse kogu tabeli läbivaatamisel sisemiselt andmete otsimiseks protseduuri *table scan*, mis tähendab kõigi tabeli ridade läbivaatamist ilma indeksit kasutamata. Mitte-klasterdatud indeksiga tabeli puhul jääb andmebaasisüsteemi enda otsustada, kas päringu täitmiseks on optimaalsem *index scan* või *table Scan* operatsioon. Kui indeksit kasutatakse üksikute ridade leidmiseks (st tervet indeksit ei ole vaja läbi vaadata), siis on selle protseduuri nimeks *index seek.* Antud teadmine tuleb kasuks (Microsoft SQL Server) täitmisplaanide lugemisel.

4.1.3 Veerupõhise indeksi realisatsioon Microsoft SQL Server'is



Joonis 10. Microsoft SQL Server veerupõhise indeksi mõistekaart

Luues veerupõhise indeksi (CSI), toimub andmete salvestamine ja lugemine veerupõhiselt. Veerupõhist indeksit hoitakse võimalusel täielikult mälus. Kogu indeksi mälus hoidmisele aitab kaasa asjaolu, et andmete salvestamine veerupõhiselt võimaldab suurel määral andmete pakkimist. SQL Server toetab nii klasterdatud kui ka mitte-klasterdatud veerupõhist indeksit (vt joonis 10). Nii klasterdatud kui ka mitte-klasterdatud veerupõhine indeks töötavad samal põhimõttel, kuid neil on siiski mõningaid erinevused. Üheks erinevuseks on, et mitte-klasterdatud veerupõhine indeks on ainult loetav e andmete muutmisel tuleb see uuesti ehitada. Teiseks erinevuseks on, et klasterdatud veerupõhine indeks ei võimalda enda kõrvale (samasse tabelisse) luua ühtegi teist (tavalist) mitte-klasterdatud indeksit.

CSI sobib väga hästi kasutamiseks OLAP (*OnLine Analytical Processing*) operatsioonide jaoks. Sinna alla kuuluvad lugemis-päringud (*read-only queries*), mis töötlevad suurt hulka andmeid. Teisisõnu, antud indeksist on oodata olulist kasu koondandmete leidmise päringutel, mitte konkreetsete olemite ja seoste andmete otsimisel (tüüpiline päring tehingutöötluse süsteemides). Koondandmete leidmise päringute täitmiseks on süsteemil tavaliselt vaja teostada ridade järjestikust läbivaatamist (*tabel scan*), mis tähendab, et süsteem vaatab läbi kõik selle tabeli read (analoogia raamatu sisu osa otsast lõpuni läbilugemisega).

CSI kasutamine toob lisaks võimalikule kiiruse kasvule kaasa ka tabeli poolt kasutatava andmemahu vähenemise. Põhjus seisneb selles, et kuna veerus olevad andmed on tihti homogeensed (ühtlased, samasugused, ühesuguste omadustega), siis annavad ka pakkimise ja kodeerimise algoritmid head tulemust. Microsoft ise lubab CSI kasutamise korral kümnekordset päringute kiiruse kasvu (OLAP päringu tüüpide korral) ja kuni seitsmekordset andmemahu vähenemist. (Columnstore Indexes Described, 2015)

4.1.3.1 Andmete pakkimine veerupõhise indeksi kasutamise korral

Andmete pakkimine vähendab nii kettal kasutatavat ruumi kui ka andmete lugemise aegu. Pakkimine on kolme-sammuline. Esiteks veeru väärtused kodeeritakse, teiseks valitakse optimaalne ridade järjekord ning kolmandaks kodeeritud väärtused pakitakse.

Kodeerimise käigus tehakse kõikidest väärtustest 32 või 64 bitised täisarvud. Selleks kasutatakse kas sõnastiku-põhist või väärtuse-põhist kodeerimist. Sõnastiku puhul salvestatakse unikaalsed väärtused massiivi, kus igal väärtusel on oma täisarvuline identifikaator, mida kasutatakse veerus väärtuse asemel. Sõnastik ise salvestatakse BLOB'ina kasutades andmebaasisüsteemi standardseid meetodeid. Väärtuse-põhist kodeerimist kasutatakse täisarvude ja kümnendmurdude korral. Selle meetodi korral leitakse iga veergude grupi jaoks eksponent ja baas. Täisarvude puhul on eksponent negatiivne, kümnendmurdude korral positiivne. Baas on väikseim väärtust peale eksponendi rakendamist. Lõpuks tuleb igast väärtusest maha lahutada baas. Meetodi eesmärgiks on mahutada veergude väärtused väiksemasse väärtuste piirkonda. Näiteks väärtuste 500, 1700 ja 1333000 korral oleks eksponent -2, sest sellega säilitatakse veel kõikide arvude täpsus. Eksponendi rakendamisel tekib rida $500*10^{-2}=5$, $1700*10^{-2}=17$, $1333000*10^{-2}=13330$. Uues reas (5, 17 ja 13330) on väikseim arv 5, seega saab see antud rea baasiks. Järgnevalt tuleb igast uue rea elemendist lahutada baas, siis saame lõpliku rea 0, 12 ja 13325. Antud rida salvestatakse maha koos baasi

ja eksponendiga. Kuna antud rea väärtuste vahemik on oluliselt väiksem (1333000-500=1332500 vs 13325-0=13325), siis kulub ka nende salvestamiseks vähem bitte.

Üks võimalik algoritm andmemahu vähendamiseks e pakkimiseks on RLE (*run-length encoding*). RLE annab parima tulemuse, kui paljude järjestikuste ridade korral on veerus ühesugused väärtused. Siis saab süsteem sisemiselt salvestada väärtuse ja selle, mitmes järgnevas reas see esineb. Kuna ühesuguse väärtusega ridade grupis ei ole ridade järjekord oluline, saab järjekorda muuta, et saavutada parim üleüldine andmete pakkimine. Kui tabelis oleks ainult üks veerg on olukord lihtne (sorteerida tuleb selle veeru järgi), kui aga veerge on rohkem kui üks tuleb rakendada keerulisemat algoritmi, et saavutada parim tulemus tabeli kui terviku jaoks. Microsoft SQL Server kasutab ridade sorteerimiseks patenteeritud algoritmi Vertipaq™. Sorteerimine on tähtis, sest RLE meetod võib ka andmemahtu kasvatada, kui read on järjestatud nii, et veerus on vähe üksteisele järgnevaid väärtuseid. (Larson, et al., 2011)

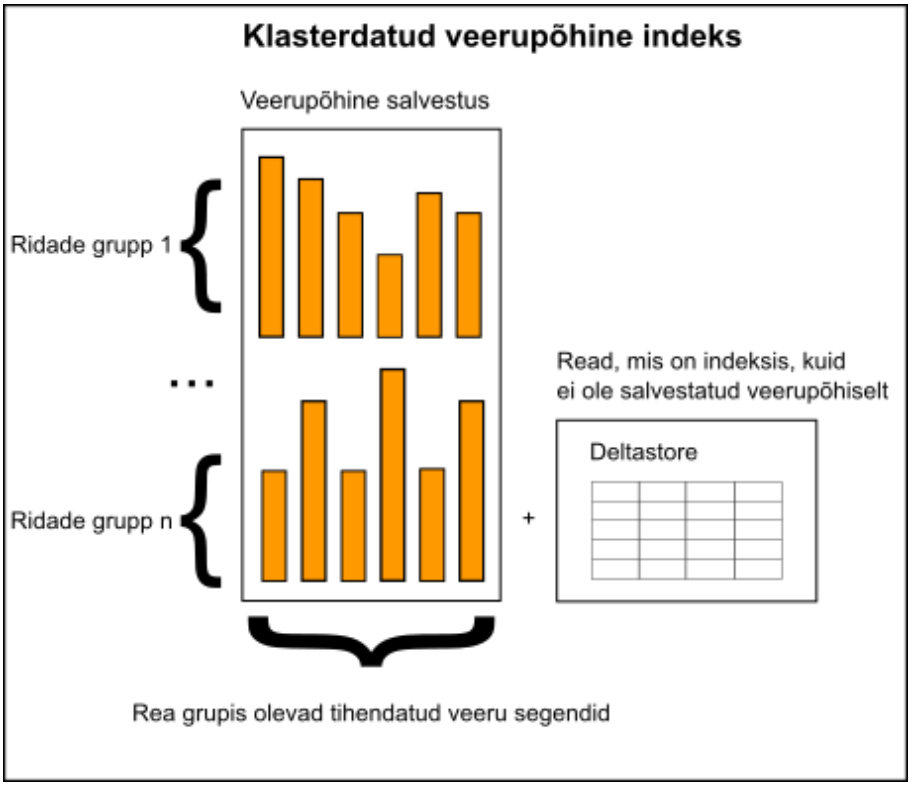
4.1.3.2 Mitte-klasterdatud veerupõhine indeks

Mitte-klasterdatud veerupõhist indeksit saab luua tavalise reapõhise salvestusega tabeli kõrvale, selle ühele või mitmele veerule. Mitte-klasterdatud indeksis on dubleeritud alamhulk indekseeritud tabeli ridadest ja veergudest. See tähendab, et mitte-klasterdatud veerupõhine indeks suurendab andmebaasi andmemahtu. Lisaks antud indeks on mõeldud ainult lugemiseks (*read-only*), mis omakorda tähendab, et tabelis ei saa andmeid ilma indeksit maha võtmata muuta. Nii `INSERT`, `UPDATE`, `DELETE` kui ka `MERGE` operatsioonid mitte-klasterdatud veerupõhise indeksiga tabeli peal annavad vea. Antud indeksi positiivseks küljeks on, et seda saab luua teiste indeksite ja kitsenduste kõrvale. (Columnstore Indexes Described, 2015)

4.1.3.3 Klasterdatud veerupõhine indeks

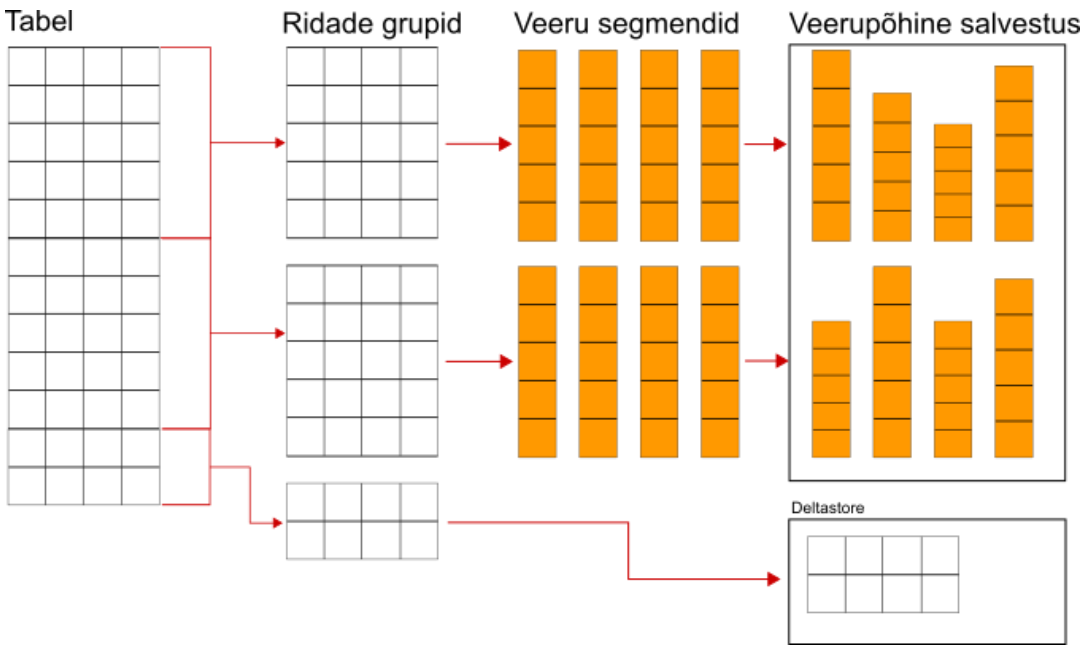
Klasterdatud veerupõhine indeks (edaspidi CCSI - *clustered columnstore index*) sobib hästi kasutamiseks andmeaitade ja suurte tabelite korral. Põhjus seisneb selles, et erinevalt mitte-klasterdatud veerupõhise indeksiga on seda tüüpi indeks ka uuendatav, st kui muuta andmeid tabelis (`INSERT, UPDATE, DELETE` ja `MERGE`), muudetakse andmeid ka sellele loodud indeksis. CCSI puhul ei looda indekseeritava tabeli kõrvale täiendavat indeksit, vaid hoopis terve tabel salvestatakse antud indeksi struktuuris (nagu teiste klasterdatud indeksite puhul). CCSI'sse kuuluvad kõik tabeli veerud ning kui selline indeks on tabelile loodud, siis teisi indekseid (sh ka kitsendusi) sellele tabelile lisada ei saa. Andmed on füüsiliselt salvestatud

selliselt, et pakkuda maksimaalset andmete pakkimist ja päringute jõudlust. Sarnaselt *C-Store* WS'iga on Microsoft SQL Serveris realiseeritud *Deltastore* (vt joonis 11), kuhu lähevad sisestatud read ning lisaks on CCSI's ka üks B-puu indeks, kus hoitakse kustutatud ridade ID'sid. Andmeid *Deltastore*'is hoitakse ajutiselt ja kõik read viiakse üle veerupõhisesse salvestusse kui nende arv ületab kindla piiri (102400 rida) (*C-Store* puhul teostas seda TM). *Deltastore* ise salvestab andmeid reapõhiselt.



Joonis 11. Microsoft SQL Serveri veerupõhise indeksi füüsiline struktuur
(Columnstore Indexes Described, 2015)

Klasterdatud indeksi loomisel tabelile lõhutakse kõigepealt tabel sisemiselt ühesuurustesse (ühesuguse ridade arvuga) ridade gruppidesse. Kõik read mis üle jäävad lisatakse *Deltastore*'i. Ridade arv grupis on minimaalselt 102400. Kõik sisestatud read jäävad *Deltastore*'i seniks, kui ületatakse antud lävend. Kui lävend ületatakse salvestatakse ka need veerupõhiselt. Reagrupid tükeldatakse veergude haaval ja nii moodustuvad veeru segmendid. Veeru segmendid pakitakse ja lisatakse veerupõhisesse salvestusse (vt joonis 12). (Columnstore Indexes Described, 2015)



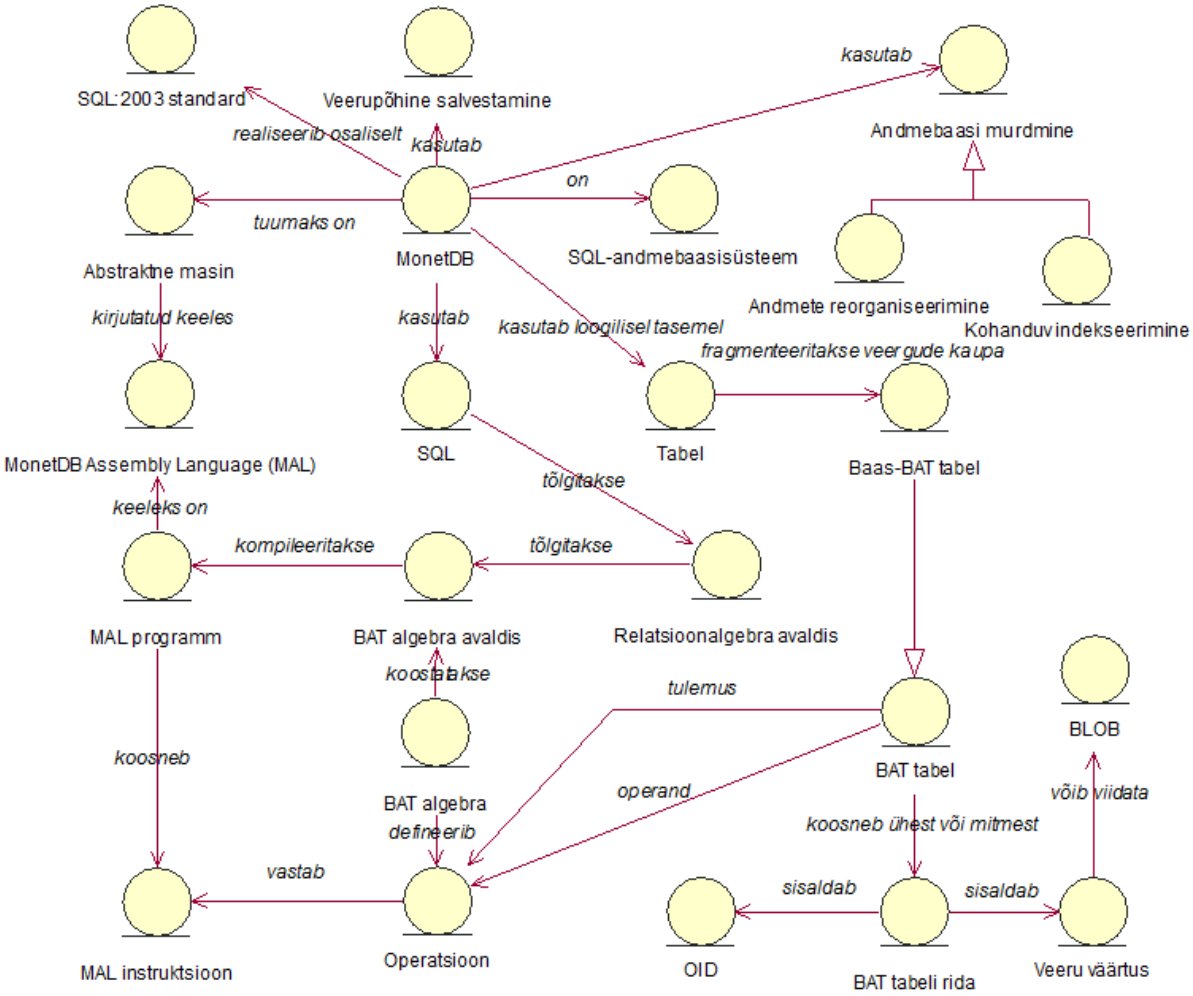
Joonis 12. Klasterdatud veerupõhise indeksi loomine Microsoft SQL Serveris (Columnstore Indexes Described, 2015)

4.2 MonetDB

Töös kasutatud versioon: 20150123, x86-64

Andmebaasikeel: SQL

Andmete salvestamise meetod: veerupõhine



Joonis 13. MonetDB mõistekaart

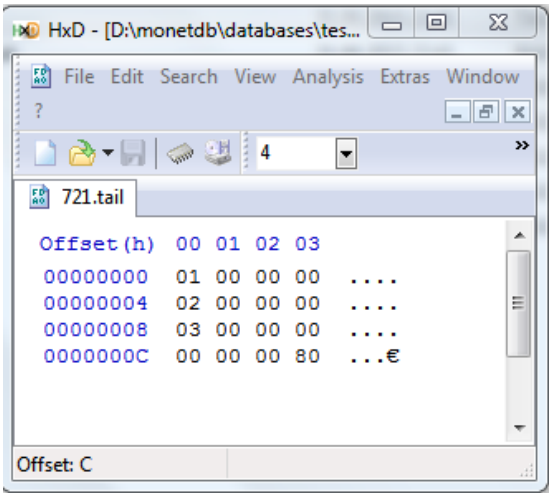
MonetDB on SQL-andmebaasisüsteem (vt joonis 13), mis järgib *SQL:2003* standardit. MonetDB andmebaasisüsteemi kasutatakse eelkõige tervishoiu, telekommunikatsiooni ja teaduse valdkonnas. Erinevalt Microsoft SQL Server andmebaasisüsteemist on see puhas veerupõhise salvestamisega andmebaasisüsteem, st see on loodud antud salvestamise meetodit silmas pidades, mitte pole sellist võimalust mingil süsteemi eluetapil juurde lisatud. MonetDB sai valitud, kuna tegemist on (erinevalt Microsoft SQL Serverist) avatud lähtekoodiga ning tasuta pakutava andmebaasisüsteemiga ning seda on kõige rohkem mainitud erinevates teadusartiklites, mis käsitlevad veerupõhise salvestamisega andmebaasisüsteeme [(Abadi, et

al., 2008), (Boncz, et al., 2005), (Harizopoulos, et al., 2006), (Ideros, et al., 2012)]. Kõigele lisaks kasutab see andmebaasisüsteem SQL keelt, mis teeb testide sooritamise lihtsamaks ja paremini võrreldavaks.

MonetDB andmebaasisüsteemi saab tasuta alla laadida nende kodulehelt (https://www.monetdb.org/Downloads). Andmebaasisüsteemiga suhtlemiseks ja SQL lausete käivitamiseks kasutasin Windows'i käsureal põhinevat programmi *mclient*.

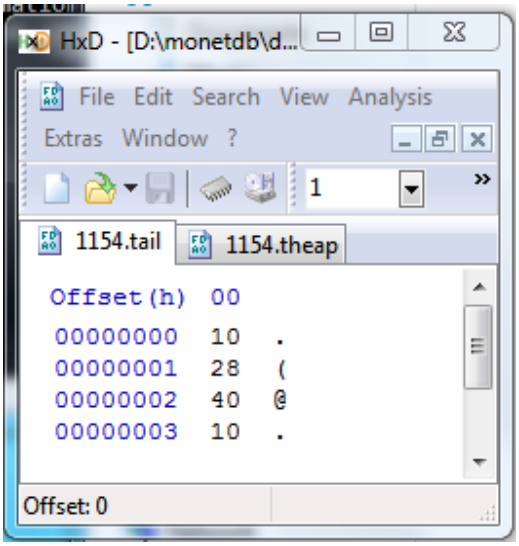
4.2.1 MonetDB salvestusmudel

Kõik tabelid on MonetDB andmebaasisüsteemis sisemisel tasemel vertikaalselt fragmenteeritud e killustatud. Iga veerg on salvestatud eraldi BAT (*Binary Association Table*) tabelisse. BAT tabel sisaldab endas alati kahte veergu ja on esitatud kujul {(surrogaat, väärtus)}. Vasakpoolset veergu tuntakse rea OID (*Object Identifier*) või päise (*head*) nime all. Parempoolset veergu, kus hoitakse reaalseid väärtuseid, tuntakse sabana (*tail*). Kui algtabelis on *k* veergu, siis moodustatakse kokku *k* BAT tabelit. Algtabelitest loodud BAT tabeleid kutsutakse ka baas-BAT tabeliteks. Algtabeli igale reale vastab OID ja see lisatakse igasse BAT tabelisse vastavalt sellele, millise rea juurde antud veeru väärtus kuulus. BAT tabelid on üldjuhul realiseeritud C-keele tüübitud massiividena. BAT tabelite puhul ei ole OID väärtuse füüsiline loomine isegi vajalik, sest seda saab tuletada veeru väärtuse asukoha (massiivi indeksi) järgi. Iga veerus olev väärtus salvestatakse BAT tabelis sama rea positsioonis (massiivi asukoha indeks on rea number) nagu see oli algtabelis. MonetDB salvestab BAT failid kõvakettale laiendiga *.bat, Võtame vaatluse alla näiteks veeru, kus salvestatakse täisarve ning NULL (väärtuse puudumist tähistav marker) on lubatud. Sisestame sinna testimiseks väärtused 1, 2, 3 ja NULL. Vaadates sellele veerule loodud BAT faili sisu programmiga *HxD*, näeme, et väärtused jooksevad järjest nelja baidi kaupa (vt joonis 14). (Ideros, et al., 2012)

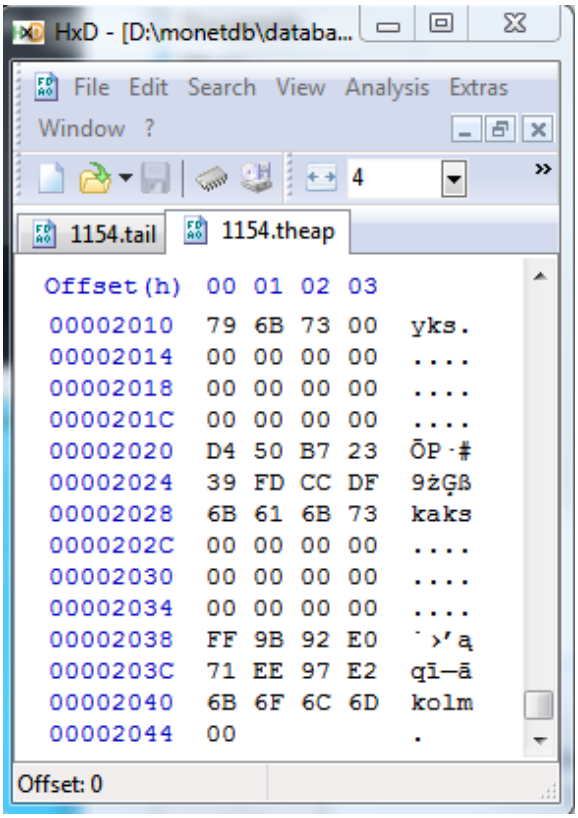


Joonis 14. MonetDB BAT tabel kõvakettal

Fikseeritud suurusega andmetüüpide korral nagu täisarv (*integer*) kasutab MonetDB C-keele vastava tüübiga massiive. Muutuva pikkusega andmetüüpide korral nagu *text* liidetakse kõik erinevad veeru väärtused kokku ja salvestatakse see BLOB'ina (*Binary Large OBject*). BAT tabelisse salvestatakse väärtuste asemel hoopis viide BLOB'i, kus konkreetne väärtus algab. Võtame vaatluse alla veeru, mille tüübiks on *text* ning kus asub neli väärtust 'yks', 'kaks', 'kolm' ja 'yks'. Sellele veerule vastavas BAT tabelis (vt joonis 15) on kirjas üksnes viited BLOB'i. Viidatavas BLOB'is (fail kõvakettal laiendiga *.theap) on näha viidatud väärtused (vt joonis 16).



Joonis 15. BAT tabel kõvakettal muutuva suurusega andmetüübi korral



Joonis 16. Muutuva suurusega andmetüübi väärtuseid hoidev BLOB

Sisemiselt käsitleb MonetDB BAT tabeleid mälus-kaardistatud failidena (*memory-mapped file*). See tähendab, et BAT failid salvestatakse kettale selliselt nagu nad on mälus. See omakorda tähendab, et MonetDB ei kuluta lisaaega failide kodeerimise ja dekodeerimise peale, mis oleks vajalik juhul kui kettale salvestatud failid oleksid teises formaadis.

Veerupõhise salvestamise meetodi üheks negatiivseks küljeks on vajadus veerge ühendada, et terviklik tabel kokku panna. Antud tegevus võib üldisele andmebaasisüsteemi jõudlusele väga halvasti mõjuda. MonetDB võitleb antud probleemi vastu kasutades hilist rea rekonstrueerimist (*late tuple reconstruction*). See tähendab, et päringu töötlemisel kasutatakse läbivalt veerupõhist formaati ning alles siis, kui vastus on vaja kasutajale esitada, rekonstrueeritakse tabel.

MonetDB tugineb madalatasemelisele BAT algebrale (relatsioonialgebra BAT tabelitel). BAT algebras defineeritud operatsioonide sisendiks on BAT tabelid ja skalaarsed suurused ning väljundiks BAT tabel. SQL päringu töötlemise tulemuseks BAT tabelite kollektsioon.

Kasutajale siiski kuvatakse talle tuttavat (loogilist) tabelit, mis pannakse kokku selle kollektsiooni põhjal.

Järgnevalt on toodud loetelu põhilistest uuenduslikest tehnoloogiatest, mis on MonetDB andmebaasisüsteemis realiseeritud.

- **Riistvara-teadlik andmebaasitehnoloogia**. Üks olulisemaid innovatsioone MonetDB andmebaasisüsteemis on, et selles kasutatakse riistvara-teadlike algoritme. Kuna protsessorite kiirused on kasvanud kiiremini kui muutmälude kiirused, siis on tekkinud pudelikael muutmälu juurde. See tähendab, et protsessor ootab kaua muutmälu järgi. Antud probleemi tuntakse ka mälu-seina (*memory wall*) nime all. Üks viis, kuidas selle vastu võidelda, on kujundada oma algoritmid ja andmestruktuurid nii, et toimuks võimalikult vähe pöördumisi muutmälu poole. Teisisõnu tuleks võimalikult palju ja võimalikult pikaks ajaks tuua kõik vajalikud andmed protsessori vahemällu.

- **Töötamine hulkadega ja kerge andmete pakkimine.** MonetDB opereerib sisemiselt mitte üksikute elementide, vaid hulkade (vektorite) kaupa. Opereerides hulkade kaupa on võimalik amortiseerida funktsiooni väljakutsumise (*function overhead*) kulusid. Hulga suurus peaks olema samas piisavalt väike, et see mahuks ära protsessori vahemällu. Lisaks hulkade kaupa opereerimise kasutab MonetDB ka kerget ja õigeaegset (*just-in-time*) andmete pakkimist, mis omakorda aitab kaasa mälu-seina probleemi leevendamisele.

- **Vahepealsete tulemuste taaskasutamine (recycling).** MonetDB andmebaasisüsteemis päringute (`SELECT` lausete) töötlemise käigus loodud vahepealseid tulemusi (BAT algebra väljundiks olevad BAT tabelid) ei kustutata, vaid salvestatakse taaskasutamiseks. Kui uue päringu täitmiseks eksisteerib sobilik ajakohane vahetulemus, siis kasutatakse seda ja ei arvutata seda uuesti. Siit järeldub, et võib tekkida olukordi, kus baas-BAT tabeleid päring ei puudutagi. Vahepealseid tulemusi hoitakse alles seni, kuni nende jaoks on veel ruumi või kuni selles olevad andmed aeguvad (andmebaasis sooritatakse andmemuudatuse lause).

- **Andmebaasi murdmine (*database cracking*).** Tänapäevaste ärirakenduste või teaduslike andmebaaside puhul võib eksisteerida keeruline olukord, kus esiteks ei ole võimalik andmebaasi ajutiselt kinni panna, et teostada selles vajalikke füüsilise disaini

> muudatusi ja teiseks ei ole võimalik ette ennustada andmebaasi töökoormust ja selles teostavaid päringuid. Teadmata eelnimetatud kriteeriume ei ole võimalik ka optimeerimine näiteks sobilike indeksite näol. MonetDB andmebaasisüsteem kasutab selle probleemi lahendamiseks tehnoloogiat nimega *database cracking*, mis käigupealt reorganiseerib füüsilisi andmeid ja loob ning muudab indekseid (kohanduvad indeksid). Kohanduvaid indekseid luuakse tükkhaaval `select`, `join` ja `projection` operaatorite osana (rakendamise ajal).

(Ideros, et al., 2012)

MonetDB andmebaasisüsteem kasutab (sisemiselt) päringute täitmiseks MAL (*MonetDB Assembly Language*) nimelist keelt. Üldkujul toimub SQL lause töötlemine järgmiselt. Kõigepealt koostatakse kasutaja SQL päringust täitmisplaan, milles on kirjeldatud relatsioonialgebra operatsioonid ja nende teostamise järjekord. Seejärel tõlgitakse loodud plaan BAT algebra kujule. Järgmiseks kompileeritakse BAT algebra MAL keelde. Lõpuks läheb see MAL programm juba MonetDB tuuma täitmisele. Nende etappide vahel toimuvad ka mitmed päringu optimiseerimise etapid.

5. Varasemad uuringud

Arvutisüsteemidel põhinevate infosüsteemide ja andmebaaside kasutuselevõtt alates 20nda sajandi teisest poolest tekitas hea ja lihtsa võimaluse ettevõtte tegevuse, teadusliku eksperimendi või mõne muu valdkonna protsesside ning tulemuste analüüsiks. Üha kiirenevalt kasvav andmehulk tekitas olukorra, kus traditsioonilised (reapõhised) andmebaasisüsteemid ei olnud enam piisavalt efektiivsed analüütiliste päringute (OLAP operatsioonide) läbiviimiseks. Veerupõhise salvestamise meetod on üks võimalus, kuidas andmebaasisüsteemide efektiivsust, analüütiliste päringute korral, parandada. Eksperimentide kohta, mis uurivad veerupõhist andmete salvestamist andmebaasisüsteemides ja testivad neis andmetöötluse operatsioonide jõudlust ning ka andmebaaside andmemahte, leidsin järgmised artiklid. Otsisin artikleid Google Scholar abil, kasutades järgnevaid otsingustringe: *c-store*, *column store* ja *column stores vs row stores*. Nendele otsingustringidele vastas väga palju erinevaid tulemusi, kuna kõikide artiklite läbitöötamine oleks olnud magistritöö mahtu arvestades ebapraktiline, siis valisin välja artiklid, mis sobisid antud töö teemaga kõige paremini kokku.

- Stonebraker, M., Abadi, J. D., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madde, S., O'Neil, E., O'Neil, P., Rasin, A., Tran, N., Zdonik, S. (2005). C-Store: A Column-oriented DBMS. VLDB '05 Proceedings, 553-564.

Antud töös tutvustati veerupõhise salvestamisega andmebaasisüsteemi prototüüpi *C-Store*. Seda andmebaasisüsteemi võrreldi kahe populaarse kommertsliku SQL-andmebaasisüsteemi vastu (nimesid töös ei täpsustatud). Üks andmebaasisüsteemidest kasutas reapõhist ja teine veerupõhist salvestamist. Mõlemas andmebaasisüsteemis loodavale andmebaasile kehtestati üks ja sama andmemahu piirang. Testimiseks loodi seitse `SELECT` lauset, mis kõik sisaldasid kokkuvõttefunktsioone.

Töö tulemused näitasid, et *C-Store* andmebaasi andmemaht oli 60% väiksem kui reapõhise salvestamisega andmebaasisüsteemis oleval andmebaasil ja 25% väiksem kui teises veerupõhise salvestamisega andmebaasisüsteemis oleval andmebaasil. Hoides kinni andmebaasidele pandud andmemahu piirangust oli *C-Store* seitsme päringu peale keskmiselt 164 korda kiirem kui reapõhise salvestamisega ja 21 korda kiirem kui teine veerupõhise

salvestamisega andmebaasisüsteem. Andmemahu piirangust loobudes olid arvud vastavalt 6,4 korda ja 6 korda *C-Store* kasuks.

- Lamb, A., Fuller, M., Varadarajan, R., Tran, N., Vandiver, B., Doshi, L., Bear, C. (2012). The Vertica Analytic Database: C-Store 7 Years Later. - Proceedings of the VLDB Endowment, Volume 5 Issue 12, 1790-1801.

Antud töös uuriti andmebaasisüsteemi *Vertica*, mis on andmebaasisüsteemi prototüübist *C-Store* väljakasvanud kommertslik andmebaasisüsteem. Eksperimendi osas võrreldi *C-Store* ja *Vertica* abil loodud andmebaasi andmemahtusid ja ka seitsme päringu kiirust.

Töö tulemused näitasid, et *Vertica* andmebaasisüsteemis loodud andmebaasi andmemaht oli üle 50% väiksem ja päringud töötasid *Vertica* andmebaasis keskmiselt kaks korda kiiremini kui *C-Store* andmebaasisüsteemi andmebaasis.

- Larson, P., Clinciu, C., Hanson N. E., Oks, A., Price, L. S., Rangarajan, S., Surna, A., Zhou, Q. (2011). SQL Server Column Store Indexes. – SIGMOD ’11, 1177-1184.

Antud töös uuriti Microsoft SQL Server 11 „Denali“ (2012) versioonis sisse toodud uut tüüpi indeksit, mis võimaldab veerupõhist salvestamist. Eksperimendis kasutatakse kuute erinevat andmebaasi. Andmebaaside disain ja nendes olevad andmed pärinesid Microsofti enda reaalsete teenuste infosüsteemidest. Igast andmebaasist loodi omakorda kaks varianti, millest ühel oli veerupõhine indeks ja teisel ei olnud. Nendes andmebaasides sooritati neli erinevat `SELECT` päringut. Esimene neist sisaldas ühte kokkuvõttefunktsiooni, teine samuti ühte kokkuvõttefunktsiooni, kuid oli väga piirava predikaadiga, kolmas oli keerulisem ja sisaldas nelja kokkuvõttefunktsiooni ja neljas sisaldas peale kokkuvõttefunktsioonide ka alampäringuid.

Töö tulemused näitasid, et maksimaalselt vähenes andmebaasi andmemaht 14,7 korda ning minimaalselt neli korda. Kõik päringud peale ühe olid veerupõhise indeksiga andmebaasis kiiremad.

- Abadi, J. D., Madden, R. S., Nachem, N. (2008). Column-Stores vs. Row-Stores: How Different Are They Really? – SIGMOD ’08, 967-980.

Antud töös võrreldi *C-Store* ja ühte reapõhise salvestusega andmebaasisüsteemi. Nendes loodi andmebaasid vastavalt *Star Schema Benchmark* (SSBM) spetsifikatsioonile. Andmebaasi keskeses tabelis oli 60 miljonit rida. Nendes andmebaasides viidi läbi 13 erinevat päringut.

Töö tulemused näitasid, et proovides emuleerida veerupõhiseid andmebaase reapõhise salvestamisega andmebaasisüsteemis ei anna häid tulemusi. Parima tulemuse reapõhiste andmebaasisüsteemide korral annab materialiseeritud vaadete e hetktõmmiste kasutamine. Siiski ka materialiseeritud vaateid kasutades oli reapõhise salvestamisega andmebaasisüsteemis läbiviidud päringud keskmiselt ligi viis korda aeglasemad kui *C-Store* andmebaasisüsteemis.

- Harizopoulos, S., Liang, V., Abadi, J. D., Madden, S. (2006). Performance Tradeoffs in Read-Optimized Databases. – VLDB ’06 Proceedings, 487-498.

Antud töö eesmärgiks oli näidata veerupõhise ja reapõhise andmete salvestamise meetodi erinevusi. Selleks loodi nullist üks veerupõhine ja üks reapõhine andmebaasisüsteem, kus realiseeriti teatud relatsioonilise andmebaasisüsteemi operatsioonid. Mõlemas süsteemis loodi üks andmebaas, kus oli kaks tabelit, kummaski 60 miljonit rida.

Töö tulemused näitasid, et veerupõhise salvestamise puhul mõjutab veeru andmetüüpidesse kuuluvate väärtuste suurus päringu täitmise kiirust, kuid reapõhise salvestamise korral mitte. Lisaks leiti, et mida „kitsamad“ veerud on tabelis (mida väiksema suurusega väärtuseid sisaldav andmetüübiga veerge kasutati), seda väiksem on vahe päringu kiiruste vahel reapõhise ja veerupõhise salvestamisega andmebaasisüsteemis. Seda sellepärast, et kettalt või muutmälust on vaja lugeda vähem andmeid e väheneb mälu-seina efekt.

Nendest uurimustest eristub käesoleva magistritöö uuring järgnevalt.

- Valikus olevad andmebaasisüsteemid.
- Uuritakse indeksite mõju veerupõhise salvestamisega andmebaasisüsteemile (MonetDB).
- Vaadeldakse ka tüüpilisi OLTP operatsioone nagu üksikute ridade otsimine, lisamine, muutmine, kustutamine.
- Uuritakse, milline on kõige efektiivsem viis paljude andmete (ridade) lisamiseks andmebaasi.
- Tulemuste abil hinnatakse erinevate kirjanduse põhjal leitud hüpoteeside paikapidavust.

6. Eksperimendi kirjeldus

6.1 Eksperimendis kasutatav riistvara

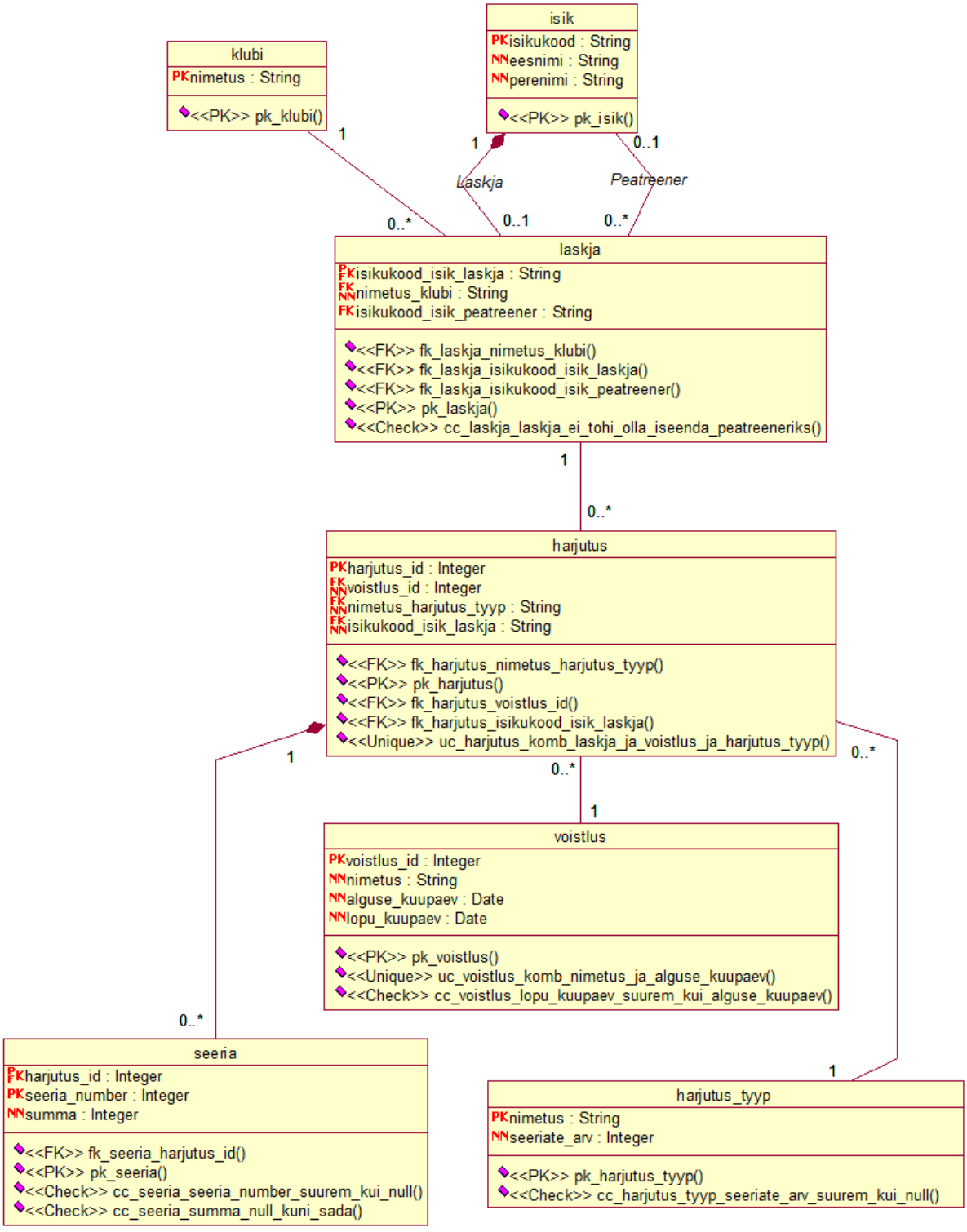
Testid viiakse läbi sülearvutil Acer Aspire 5943G.

- Operatsioonisüsteem: Windows 7® Home Premium 64-bit
- Failisüsteem: NTFS, lehekülje (klastri) suurus 4KB
- Protsessor: Intel® Core™ i7-720QM, 1,6GHz, 4-tuuma, 8-lõime, 32KB L1 taseme andmete vahemälu iga tuuma kohta, 256KB L2 taseme vahemälu iga tuuma kohta, 6MB L3 taseme vahemälu, mis on jagatud tuumade vahel, vahemälu rea suurus kõikide kolme taseme korral on 64B
- Mälu: 4GB DDR3
- Kõvaketas: WD 500GB HDD 5400 rpm, lugemiskiirus ~70MB/s, pöördusaeg 12ms (keskmine latentsus 5,5ms)

6.2 Eksperimendis kasutatavate andmebaaside loogiline mudel

Oma bakalaureusetöös (Puustusmaa, 2012) uurisin erinevat tüüpi võtmete kasutamise eeliseid ja puuduseid SQL-andmebaasis ning selle raames projekteerisin andmebaasi laskespordi võistluste tulemuste talletamiseks. Täpsemalt projekteerisin bakalaureusetöö käigus kolm andmebaasi, mis erinesid üksnes nende tabelites kasutatavate võtmete poolest. Bakalaureusetöös koostatud testide põhjal osutus kõige paremaks kombineeritud võtmetega andmebaasi disain. Seega kasutan antud töös nendest kolmest loodud disainist kombineeritud võtmetega disaini (vt joonis 17).

Andmebaasi transaktsiooniliste andmetega tabeliteks on *harjutus*, *seeria* ja *võistlus*. Ülejäänud tabelid *isik*, *klubi*, *laskja* ja *harjutus_tyyp* on põhiandmete (*master data*) või klassifikaatorite rollis ning seal tehakse andmemuudatusi võrdlemisi harva. Transaktsiooniliste andmetega tabelitest sisaldab kõige rohkem ridu tabel *seeria*, natuke vähem ridu on tabelis *harjutus* ja tabel *voistlus* sisaldab eelnevast kahest juba oluliselt vähem andmeid.



Joonis 17. Andmebaasi loogiline disain

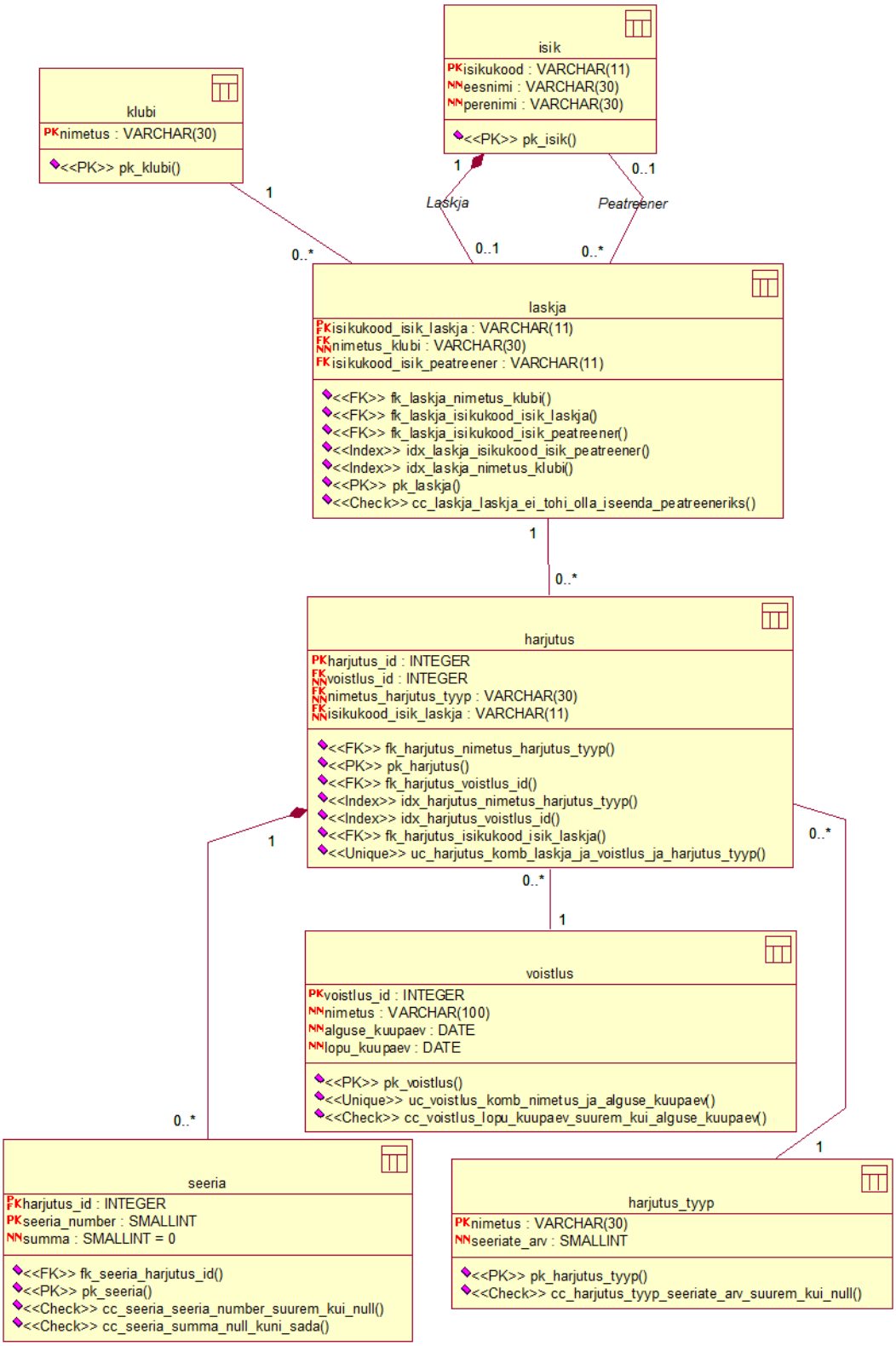
6.3 Eksperimendis kasutavate andmebaaside füüsiline disain

Oma eksperimendi käigus loon andmebaasi loogilise disaini alusel neli andmebaasi. Järgnevalt on nimetatud ka nende andmebaaside lühikesed nimed, mida edaspidistes kirjeldustest andmebaasidele viitamiseks kasutatakse.

- **MSR**: Microsoft SQL Serveri reapõhise salvestamisega andmebaas ilma veerupõhist indeksit kasutamata, kuid kasutades kõiki teisi indekseid ja kitsendusi.
- **MSV**: Microsoft SQL Serveri andmebaas kasutades veerupõhist indeksit, kuid kasutamata ühtegi teist täiendavat indeksit ja kitsendust.
- **MDBI**: MonetDB andmebaas kasutades indekseid ja kitsendusi.
- **MDB**: MonetDB andmebaas kasutamata ühtegi täiendavat indeksit ja kitsendust.

Järgnevalt kirjeldatakse iga selle andmebaasi füüsilist disaini detailsemalt, tuues täpselt välja, millised kitsendused ja indeksid sellele luuakse ning millised mitte ja miks.

6.3.1 Andmebaasi disain Microsoft SQL Server andmebaasisüsteemis ilma veerupõhist indeksit kasutamata (MSR)



Joonis 18. Andmebaasi disain MSR korral

Veerupõhiseid indekseid mitte kasutavas (reapõhist salvestamist kasutavas) Microsoft SQL Serveri andmebaasis luuakse tabelitele nii primaarvõtme, välisvõtme, unikaalsuse kui ka kontroll (`CHECK`) kitsendused (vt joonis 18). Tabelite loomise laused koos kontroll kitsendustega on ära toodud lisas 1 ja teiste kitsenduste ning indeksite loomise laused lisas 3. Indeksid luuakse kõigile välisvõtmetele, välja arvatud juhul, kui välisvõtme veerg on esimene veerg primaarvõtme või unikaalsuse kitsenduse võtmes, sest siis oskab andmebaasisüsteem kitsendust (kitsendusele loodud indeksit) kasutada. Microsoft SQL Server loob primaarvõtme kitsenduse kontrolli lihtsustamiseks automaatselt klasterdatud unikaalse indeksi.

Eelnevast tulenevalt *ei loo* ma indeksit järgnevatele välisvõtmetele.

- Tabeli *laskja* välisvõti (*isikukood_isik_laskja*)
- Tabeli *harjutus* välisvõti (*isikukood_isik_laskja*)
- Tabeli *seeria* välisvõti (*harjutus_id*)

Unikaalsuse kitsendused loon järgmistele veergudele.

- Tabel *harjutus*, veerud *isikukood_isik_laskja*, *voistlus_id* ja *nimetus_harjutus_tyyp* kitsendus *uc_harjutus_komb_laskja_ja_voistlus_ja_harjutus_tyyp*

- Tabel *voistlus*, veerud *nimetus* ja *alguse_kuupaev*, kitsendus *uc_voistlus_komb_nimetus_ja_alguse_kuupaev*

Kontroll kitsendused loodi järgmistele tabelitele (nimed peaksid sisu piisavalt selgitama).

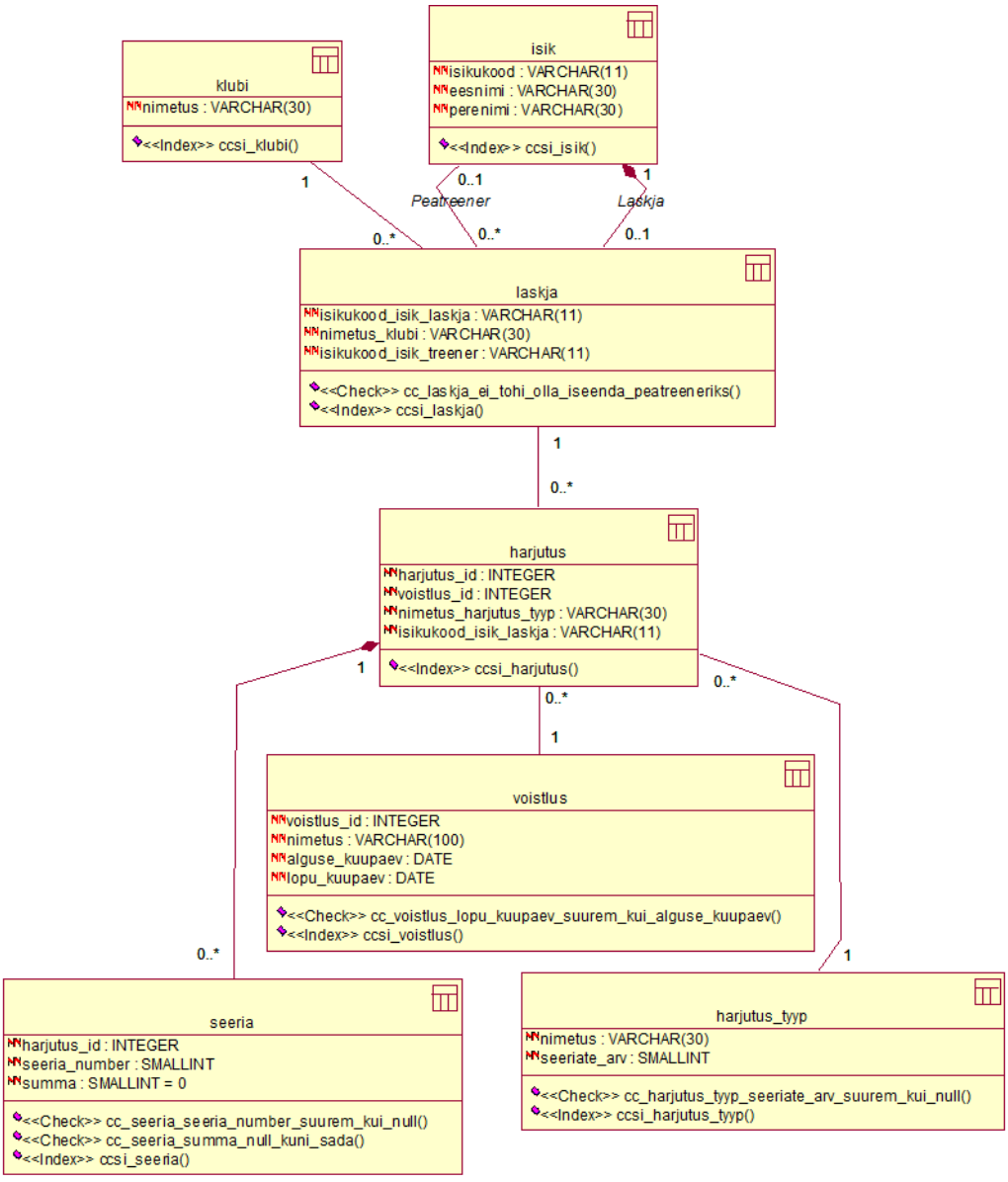
- Tabel *harjutus_tyyp*, kitsendus
  `cc_harjutus_tyyp_seeriate_arv_suurem_kui_null`

- Tabel *laskja*, kitsendus
  `cc_laskja_laskja_ei_tohi_olla_iseenda_peatreeneriks`

- Tabel *seeria*, kitsendused
  `cc_seeria_seeria_number_suurem_kui_null`
  `cc_seeria_summa_null_kuni_sada`

- Tabel *voistlus*, kitsendus
  `cc_voistlus_lopu_kuupaev_suurem_kui_alguse_kuupaev`

6.3.2 Andmebaasi disain Microsoft SQL Server andmebaasisüsteemis kasutades veerupõhist indeksit (MSV)



Joonis 19. Andmebaasi disain MSV korral

Veerupõhiseid indekseid kasutavas Microsoft SQL Server andmebaasis (vt joonis 19) loon igale tabelile ühe klasterdatud veerupõhise indeksi.

- Tabel *harjutus*, klasterdatud indeks *ccsi_harjutus*
- Tabel *harjutus_tyyp*, klasterdatud indeks *ccsi_harjutus_tyyp*
- Tabel *isik*, klasterdatud indeks *ccsi_isik*
- Tabel *klubi*, klasterdatud indeks *ccsi_klubi*

- Tabel *laskja,* klasterdatud indeks *ccsi_laskja*
- Tabel *seeria,* klasterdatud indeks *ccsi_seeria*
- Tabel *voistlus,* klasterdatud indeks *cci_voistlus*

Klasterdatud veerupõhine indeks kehtib kõikide tabeli veergude jaoks e kõik selle tabeli veerud salvestatakse klasterdatud veerupõhisesse indeksisse.

Microsoft SQL Server andmebaasisüsteem ei luba luua ühtegi teist täiendavat indeksit klasterdatud veerupõhise indeksi kõrvale. Selletõttu ei olnud tehniliselt võimalik luua järgnevaid kitsendusi.

- primaarvõtme kitsendust – kasutab kehtestamiseks unikaalset indeksit.
- unikaalsuse kitsendus – kasutab kehtestamiseks unikaalset indeksit.
- välisvõtme kitsendus – viidataval veerul peab olema primaarvõtme kitsendus, unikaalsuse kitsendus või unikaalne indeks.

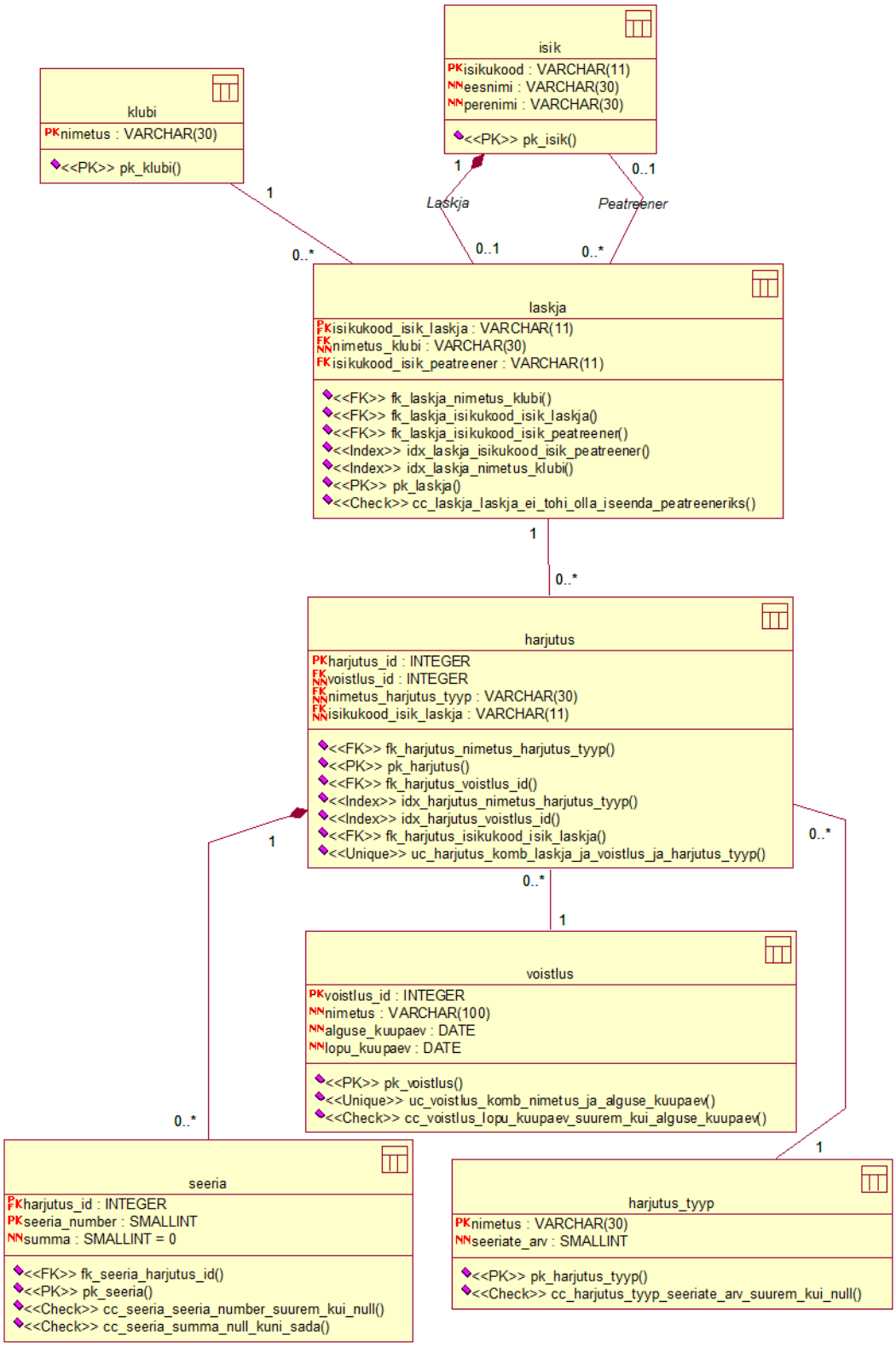
Primaarvõtme/unikaalsuse kitsenduse puudumise tulemuseks on, et põhimõtteliselt on tabelites võimalikud korduvad read. Välisvõtme kitsenduse puudumise tulemuseks on, et põhimõtteliselt võivad andmed minna vastuollu viidete terviklikkuse reegliga. Andmetega tegeleja peab sellega kindlasti arvestama.

Kontroll kitsendused loodi järgmistele tabelitele.

- Tabel *harjutus_tyyp*, kitsendus
  `cc_harjutus_tyyp_seeriate_arv_suurem_kui_null`
- Tabel *laskja,* kitsendus
  `cc_laskja_laskja_ei_tohi_olla_iseenda_peatreeneriks`
- Tabel *seeria*, kitsendused
  `cc_seeria_seeria_number_suurem_kui_null`
  `cc_seeria_summa_null_kuni_sada`
- Tabel *voistlus*, kitsendus
  `cc_voistlus_lopu_kuupaev_suurem_kui_alguse_kuupaev`

Tabeli loomise laused koos kontroll (`CHECK`) kitsendustega ja klasterdatud veerupõhiste indeksite loomise laused on ära toodud lisades (vt lisa 1 ja lisa 4).

6.3.3 Andmebaasi disain MonetDB andmebaasisüsteemis kasutades indekseid (MDBI)

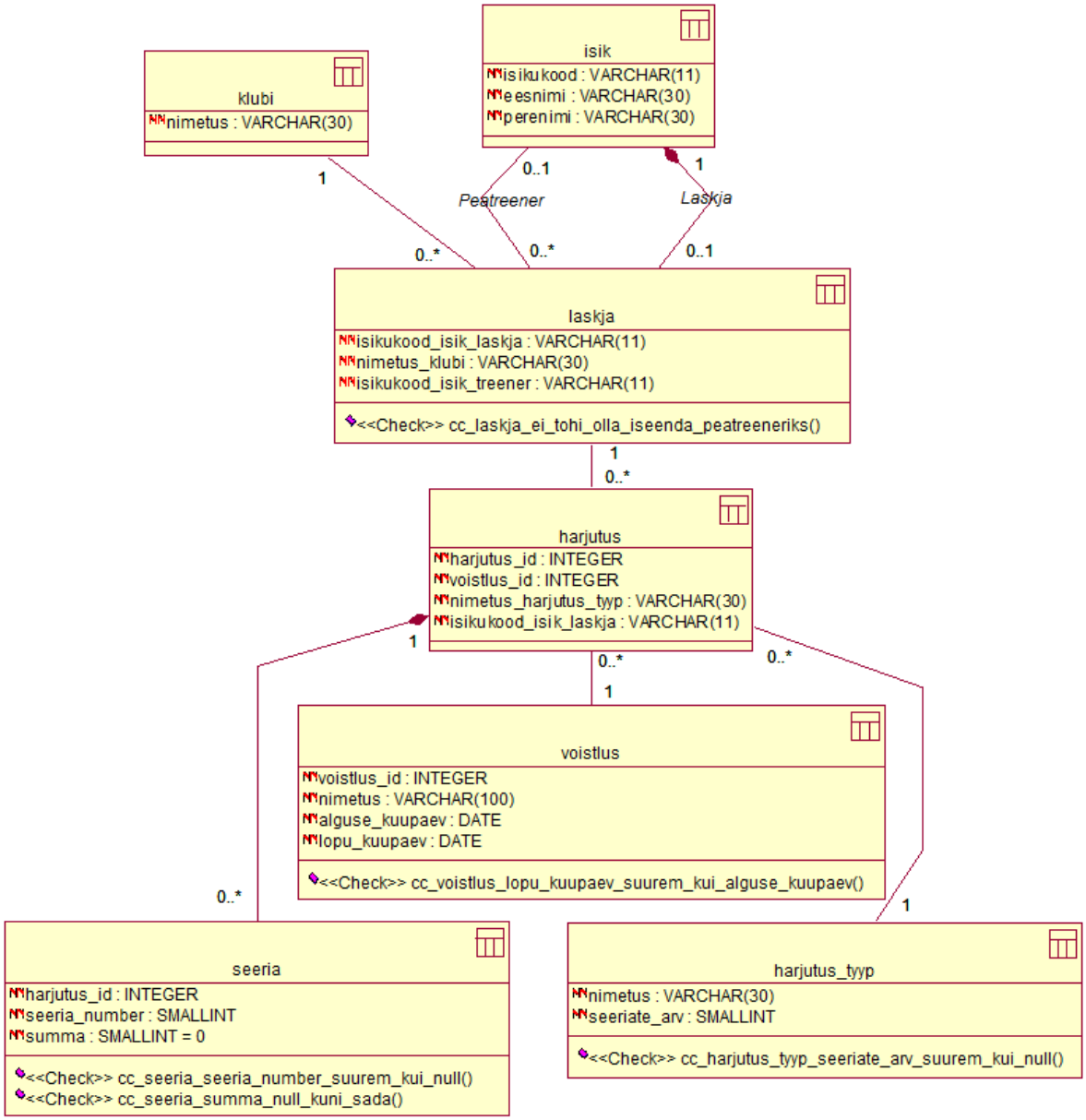


Joonis 20. Andmebaasi disain MDBI korral

Selles MonetDB andmebaasis loon kõik samad indeksid ja kitsendused (vt joonis 20), mis on Microsoft SQL Serveri reapõhise salvestamisega (MSR) andmebaasis.

Tabelite loomise laused koos kontroll (`CHECK`) kitsendustega ja indeksite ning teiste kitsenduste loomise laused on ära toodud lisades (vt lisa 2 ja lisa 3).

6.3.4 Andmebaasi disain MonetDB andmebaasisüsteemis kasutamata ühtegi täiendavat indeksit (MDB)



Joonis 21. Andmebaasi disain MDB korral

Antud andmebaas loodi selleks, et tekiks parem võrdlusmoment Microsoft SQL Serveris loodud veerupõhist salvestamist kasutava andmebaasiga (MSV). Teisisõnu, kuna MSV andmebaasis ei olnud tehniliste piirangute tõttu võimalik teatud kitsendusi ja indekseid luua, siis ei loonud ma neid ka selles MonetDB andmebaasisüsteemi andmebaasis (vt joonis 21). Sarnaselt MSV andmebaasiga lisasin tabelitele ainult CHECK kitsendused. SQL laused tabelite loomiseks koos kitsendustega on ära toodud lisades (vt lisa 2).

7. Operatsioonid

Kirjanduse järgi otsustades võiks veerupõhise salvestusega andmebaasisüsteemides kiiremini töötada päringud, mis töötlevad palju ridu aga samas kasutavad või tagastavad vähe veerge. *Päring 1* kuulub just sellesse kategooriasse, sest tabelist *harjutus* kui ka tabelist *seeria* kasutatakse 10% ridadest ning lisaks kuvatakse päringu väljundis ainult kahte veergu *isikukood_isik_laskja* ja *summa*. Seevastu *Päring 2* käsitleb kõiki võimalikke veerge. Päring ise on semantiliselt sarnane päringuga 1, kuid `WHERE` klauslisse on kaasatud kõik võimalikud veerud. Täiendavad veerud on `WHERE` klauslis selleks, et sundida veerupõhise salvestamisega andmebaasisüsteemi lugema kõiki andmelehekülgi ja näha kui palju see mõjutab päringu kiirust. Lisatud veerud on puhtalt tehnilised see tähendab, et nad ei muuda kasutatud tabelite korral päringu tulemust võrreldes päringuga 1. Muutunud päringu aeg sisaldab endas kindlasti ka loogikaoperaatorite rakendamise aegu, kuid see ei ole võrdlemisel probleemiks, sest nii on kõigis võrreldavates andmebaasisüsteemides.

Veerupõhise salvestusega andmebaasisüsteemidel ei tohiks aga olla erilist eelist, kui päringu `WHERE` klauslis olevale predikaadile vastavad üksikud read või ainult üks rida. *Päring 3* vajab täitmiseks vähe ridu, kasutades tabeli harjutus 0,1% ridadest ning *päring 4* vajab veelgi vähem, kasutades 0,01% tabeli *seeria* ridadest. Nii päring 3 kui ka 4 kasutab väljundis kahte veergu – ühte neist kuvatakse otse, teine on kokkuvõttefunktsiooni argumendiks. Viiendaks päringuks on tüüpiline operatiivandmebaasi päring, mis ei kasuta ühtegi kokkuvõttefunktsiooni ja mis otsib ühte kindlat rida (olemit) tabelist *harjutus*. (vt tabel 2)

Kuna veerupõhise salvestusega andmebaasides ei hoita ridu andmebaasi sisemisel tasemel koos, siis võib eeldada, et kõik andmete lisamise laused täidetakse aeglasemalt kui võrreldavas reapõhise salvestusega andmebaasisüsteemis. Andmete uuendamise (ridade muutmise) kiirus oleneb sellest, kuidas on see funktsionaalsus realiseeritud vastavas andmebaasisüsteemis. Kui andmete uuendamine on realiseeritud sarnaselt *C-Store'*s `INSERT` ja `DELETE` lause kombinatsioonina, siis võib eeldada, et ka see operatsioon on aeglasem veerupõhise salvestamisega andmebaasisüsteemides kui reapõhiste salvestamisega andmebaasisüsteemides. Sooritatud andmete otsimise, lisamise, uuendamise ja kustutamise SQL laused on kirjas tabelis 2. Kõik eelnimetatud tegevused toimuvad kõikides

andmebaasisüsteemides *autocommit* režiimis, st iga lauset käsitletakse eraldiseisva transaktsioonina.

Tabel 2. Operatsioonide SQL laused

Test	***Kirjeldus***
Päring 1	**Laskuri parim seeria harjutuse tüübis *HT*.**

```
SELECT harjutus.isikukood_isik_laskja, MAX(summa) AS "parim seeria"
FROM seeria
INNER JOIN harjutus ON harjutus.harjutus_id = seeria.harjutus_id
WHERE harjutus.nimetus_harjutus_tyyp = HT
GROUP BY harjutus.isikukood_isik_laskja;
```

Päring 2	**Laskuri parim seeria harjutuse tüübis *HT*.**

```
SELECT harjutus.isikukood_isik_laskja, MAX(summa) AS "parim seeria"
FROM seeria
INNER JOIN harjutus ON harjutus.harjutus_id = seeria.harjutus_id
WHERE harjutus.harjutus_id <> 0 AND
harjutus.voistlus_id <> 0 AND
harjutus.nimetus_harjutus_tyyp = HT AND
harjutus.isikukood_isik_laskja <> '' AND
seeria.harjutus_id <> 0 AND
seeria_number <> 0 AND
seeria.summa <> 0
GROUP BY harjutus.isikukood_isik_laskja;
```

Päring 3	**Aastal *A* võistlustest osa võtnud laskurite arv.**

```
--Microsoft SQL Server
SELECT YEAR(voistlus.alguse_kuupaev) AS aasta, COUNT(DISTINCT
harjutus.isikukood_isik_laskja) AS laskjaid FROM harjutus
INNER JOIN voistlus ON
harjutus.voistlus_id = voistlus.voistlus_id
WHERE YEAR(voistlus.alguse_kuupaev) = A
GROUP BY YEAR(voistlus.alguse_kuupaev);

--MonetDB
SELECT EXTRACT(YEAR FROM voistlus.alguse_kuupaev) as aasta,
COUNT(DISTINCT harjutus.isikukood_isik_laskja) AS laskjaid FROM
harjutus
INNER JOIN voistlus ON
harjutus.voistlus_id = voistlus.voistlus_id
WHERE EXTRACT(YEAR FROM voistlus.alguse_kuupaev) = A
GROUP BY aasta;
```

Päring 4	**Laskuri L rekord harjutuse tüübis HT.**

```
SELECT MAX(tulemused.tulemus) AS 'parim tulemus' FROM
      (SELECT harjutus.isikukood_isik_laskja, SUM(seeria.summa) AS
tulemus FROM seeria
      INNER JOIN harjutus ON
      harjutus.harjutus_id = seeria.harjutus_id
      WHERE harjutus.nimetus_harjutus_tyyp = HT AND
      harjutus.isikukood_isik_laskja = L
      GROUP BY seeria.harjutus_id, harjutus.isikukood_isik_laskja)
AS tulemused
GROUP BY tulemused.isikukood_isik_laskja;
```

Päring 5	**Leida harjutus H. Väljastada laskja klubi nimi, isikukood, ees- ja perenimi ühe sõnena, harjutuse identifikaator ning tüüp ja võistluse nimetus koos alguse ja lõpu ajaga.**

```
--Microsoft SQL Server
SELECT isik.isikukood + ' ' + isik.eesnimi + ' ' + isik.perenimi + '
' + laskja.nimetus_klubi AS 'laskja', harjutus.nimetus_harjutus_tyyp
AS 'harjutus', harjutus.harjutus_id, voistlus.nimetus,
voistlus.alguse_kuupaev, voistlus.lopu_kuupaev
FROM harjutus
INNER JOIN laskja on laskja.isikukood_isik_laskja =
harjutus.isikukood_isik_laskja
INNER JOIN isik on isik.isikukood = laskja.isikukood_isik_laskja
INNER JOIN voistlus on voistlus.voistlus_id = harjutus.voistlus_id
WHERE harjutus_id = H;

--MonetDB
SELECT isik.isikukood || ' ' || isik.eesnimi || ' ' || isik.perenimi
|| ' ' || laskja.nimetus_klubi AS "laskja",
harjutus.nimetus_harjutus_tyyp AS "harjutus", harjutus.harjutus_id,
voistlus.nimetus, voistlus.alguse_kuupaev, voistlus.lopu_kuupaev
FROM harjutus
INNER JOIN laskja on laskja.isikukood_isik_laskja =
harjutus.isikukood_isik_laskja
INNER JOIN isik on isik.isikukood = laskja.isikukood_isik_laskja
INNER JOIN voistlus on voistlus.voistlus_id = harjutus.voistlus_id
WHERE harjutus id = H;
```

Lisamine	**Lisame harjutusele *H* uue seeria numbriga *N* summaga *S*.**

```
INSERT INTO seeria(harjutus_id, seeria_number, summa) VALUES (H, N,
S);
```

Uuendamine	**Uuendame harjutuses *H* lastud seeria numbriga *N* väärtust asendades selle uue väärtusega *S*.**

```
UPDATE seeria SET summa = S WHERE harjutus_id = H AND seeria_number
= N;
```

Kustutamine	**Kustutame harjutuses *H* lasud seeria numbriga *N*.**

```
DELETE FROM seeria WHERE harjutus_id = H AND seeria_number = N;
```

Ainuke nõue väärtuste A, H, HT, L, S, N valikul on see, et nad peavad olema sobivat tüüpi ja eksisteerima andmebaasis.

Enne testide käivitamist ja peale andmete lisamist andmebaasi teostasin andmebaasi statistika värskendamise. Päringuid käivitati iga andmebaasisüsteemi korral kaks korda, et näha kui efektiivselt suudab andmebaasisüsteem ära kasutada muutmälu. Välja on toodud mõlema käivituskorra ajad. Esimene kord on nn külmkäivitus, kus muutmällu pole veel andmeid loetud ning ühtegi täitmisplaani koostatud. Teisel korral on juba osa või kõik lause täitmiseks vajalikud andmed muutmälus olemas ja samuti ei pea andmebaasisüsteem uuesti koostama ka täitmisplaani, sest ka see salvestatakse muutmälus.

Microsoft SQL Serveri puhul tegin külmkäivituse jaoks tühjaks puhvri (*buffer pool*). Puhver on muutmälu piirkond, kus Microsoft SQL Server salvestab päringu ajal kasutatud/loetud leheküljed. Lisaks puhastasin ka planeerija vahemälu. Täpsemalt käivitasin järgmised käsud.

```
USE laskmine;
GO
CHECKPOINT;
GO
DBCC DROPCLEANBUFFERS;
DBCC FREEPROCCACHE;
SET STATISTICS TIME ON;
SET STATISTICS IO ON;
```

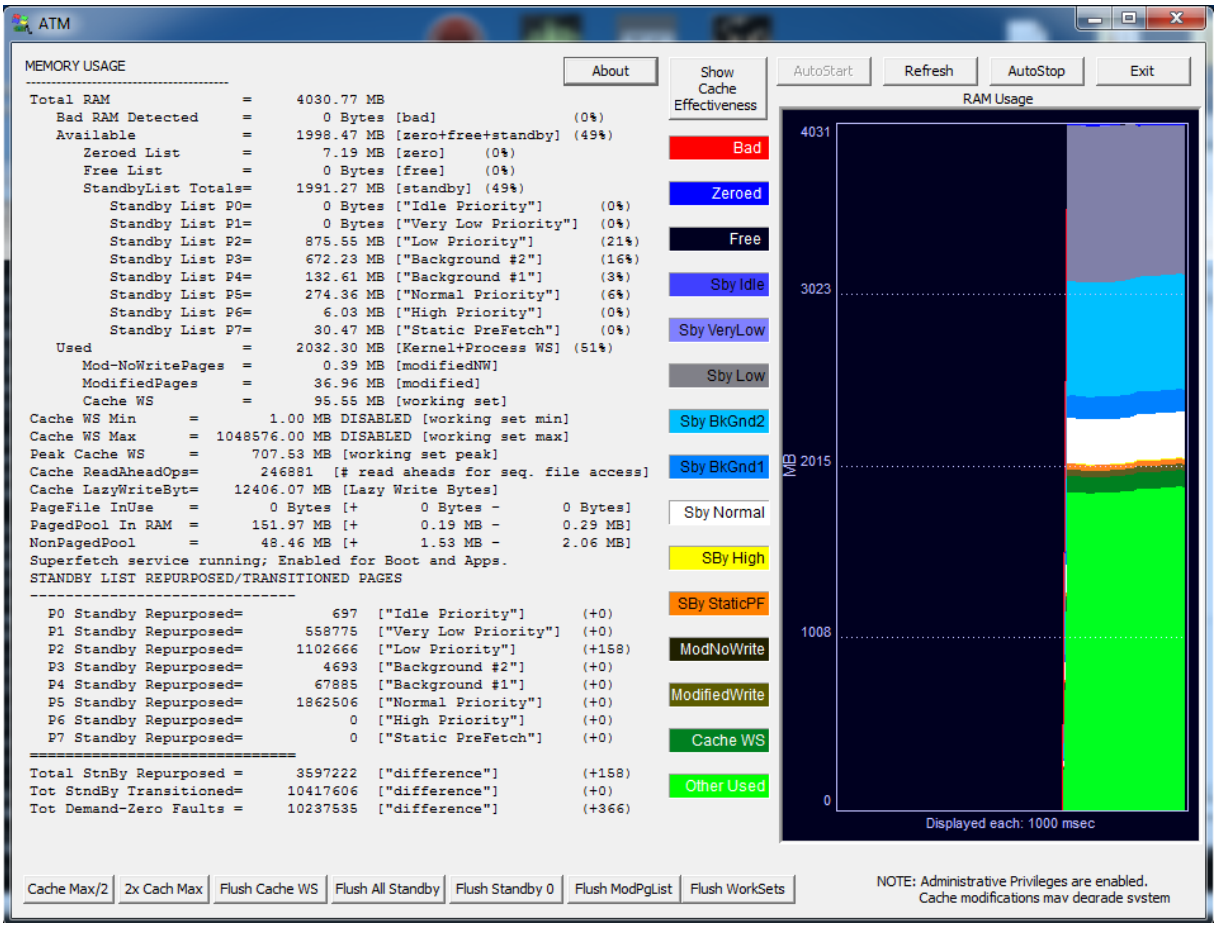
Käsuga `USE` määrame ära, millise andmebaasi kontekstis käsud käivitatakse. Käsk `CHECKPOINT` kirjutab kõik mustad leheküljed (*dirty pages*) e mälus andmetesse tehtud muudatused kettale, `DROPCLEANBUFFERS` puhastab puhvri ning `FREEPROCCACHE` puhastab planeerija vahemälu (st järgnevate päringute käivitamisel peab süsteem koostama iga päringu täitmisplaani uuesti, mitte ei saa kasutada eelnevalt koostatud ja meelde jäetud täitmisplaani).

Lausete täitmise paremaks mõistmiseks on vaja analüüsida nende täitmisplaane. Täitmisplaan on andmebaasisüsteemi poolt kokkupandud juhend, mis kirjeldab, kuidas antud deklaratiivset andmebaasikeele lauset sisemiselt protseduurina realiseerida. Täitmisplaanis on ära toodud süsteemi tegevused ja nende täitmise järjekord. Tavaliselt esitatakse täitmisplaan puustruktuurina, kus lehtedeks on lähteandmete lugemise operatsioonid ning juureks lause kui terviku täitmine. Täitmisplaan koostatakse enne, kui andmebaasisüsteem läheb andmete otsimiseks või muutmiseks reaalsete tabelite ja andmete kallale. Ühe ja sama lause täitmiseks eksisteerib mitu soovitud tulemuseni viivat täitmisplaani, mis erinevad üksteisest tavaliselt täitmisele kuluva aja poolest. Andmebaasisüsteem teeb täitmisplaani valiku vastavalt lause struktuurile ja andmetele tabelites. Ei oleks eriti efektiivne käia enne lause täitmist läbi kõik tabelid, et koguda informatsiooni nendes olevate andmete kohta. Lahenduseks on andmebaasisüsteemi poolt tabelite kohta statistika kogumine. Andmebaasisüsteem võib koguda statistikat automaatselt, kuid vastavate volitustega kasutaja võib ka anda selleks korralduse käsitsi. Alati on soovitatav peale igat suuremat andmesisestust või andmete kustutamist uuendada ka andmebaasi statistikat. Olukorras, kus andmete mahud on andmebaasis oluliselt muutunud, aga samas statistikat ei ole kaasajastatud, võib

andmebaasisüsteem koostada mitteoptimaalseid täitmisplaane. Mitteoptimaalsed täitmisplaanid tähendavad üldjuhul aeglasemat päringute täitmist.

Microsoft SQL Serveri puhul kasutan täitmisplaanide vaatamiseks *SQL Server 2014 Management Studio* tarkvara poolt pakutavat funktsionaalsust *Include Actual Execution Plan*, mille leiab *Query* menüüst. Tulemuseks on graafiline täitmisplaani esitus. Selleks, et näha summaarset päringu täitmiseks või andmete muutmiseks kulunud aega, on selle kuvamine vaja eraldi sisse lülitada. Aja kuvamise sisselülitamiseks tuleb käivitada lause `SET STATISTICS TIME ON`. Lisaks on lausete analüüsis vaja teada ka informatsiooni, kui palju andmebaasisüsteem loeb lehekülgi mälust ja kui palju kõvakettalt. Sellise informatsiooni kuvamiseks on vaja käivitada lause `SET STATISTICS IO ON`.

Esimene kord e nn külma käivituse jaoks tehti MonetDB puhul tühjaks failisüsteemi vahemälu. See tähendab, et esimesel päringu käivitamise korral loetakse kõik andmed kettalt (Query timing, 2015). Failisüsteemi vahemälu puhastamiseks kasutasin programmi ATM (ATM: The Cache Machine!, 2015) funktsionaalsusi „Flush Cache WS“ ja „Flush All Standby“ (vt joonis 22). Esimene neist puhastab failisüsteemi vahemälu (mälus-kaardistatud BAT failid asuvad seal) ning teine puhastab nn ooterežiimis oleva failisüsteemi vahemälu, kus asuvad mällu loetud failid, mida ükski protsess ei kasuta, kuid mille andmeid pole veel ülekirjutatud ja on seega valmis kiireks taaskasutamiseks. Enne nende kahe funktsionaalsuse kasutamist panin kinni nii MonetDB kliendi kui ka serveri programmi (protsessi).



Joonis 22. TMurgent ATM

MonetDB puhul kasutan täitmisplaanide vaatamiseks võtmesõna `PLAN`. Antud võtmesõna tuli lisada käivitatava päringu ette. Tulemuseks on tekstipõhine täitmisplaani esitus. Võtame näiteks järgneva lause.

```
PLAN SELECT harjutus.isikukood_isik_laskja, MAX(summa) AS
"parim seeria" FROM seeria
INNER JOIN harjutus ON harjutus.harjutus_id =
seeria.harjutus_id
WHERE
harjutus.nimetus_harjutus_tyyp = 'tyyp'
GROUP BY harjutus.isikukood_isik_laskja;
```

Järgnevas täitmisplaanis olen värvid lisanud ise, et paremini seletada, kuidas seda lugeda. MonetDB poolt loodud plaanis värve ei ole.

```
+-------------------------------------------------------------------------------+
| rel                                                                           |
+===============================================================================+
| project (                                                                     |
| | group by (                                                                  |
| | | join (                                                                    |
| | | | table(laskmine.seeria) [ seeria.summa NOT NULL, seeria.%fk_seeria       |
: _harjutus_id NOT NULL JOINIDX laskmine.seeria.fk_seeria_harjutus_id ] COUNT,:
| | | | select (                                                                |
| | | | | table(laskmine.harjutus) [ harjutus.nimetus_harjutus_tyyp NOT NULL |
: HASHCOL , harjutus.isikukood_isik_laskja NOT NULL, harjutus.%TID% NOT NULL ]:
: COU NT                                                                      :
| | | | ) [ harjutus.nimetus_harjutus_tyyp NOT NULL HASHCOL  = varchar(30) "t |
: yyp" ]                                                                      :
| | | ) [ seeria.%fk_seeria_harjutus_id NOT NULL = harjutus.%TID% NOT NULL JO |
: INIDX laskmine.seeria.fk_seeria_harjutus_id ]                               :
| | ) [ harjutus.isikukood_isik_laskja NOT NULL ] [ harjutus.isikukood_isik_l |
: askja NOT NULL, laskmine.max no nil (seeria.summa NOT NULL)                 :
: NOT NULL as L1.L1 ]                                                         :
| ) [ harjutus.isikukood_isik_laskja NOT NULL, L1 NOT NULL as L1.parim seeria |
:  ]                                                                          :
+-------------------------------------------------------------------------------+
```

Tegevused, mille taane on kõige suurem, käivitatakse kõigepealt ja nende tegevuste väljund antakse ühe taande võrra väiksema tegevuse sisendiks. Antud näite täitmisplaani puhul on tegevuste järjekord järgmine.

1. Esimene järgu tegevused (roheline) – operatsiooniga *table(laskmine.harjutus)* koostatakse vajalike veergudega harjutuse tabel.

2. Teise järgu tegevused (sinine) – siin sooritatakse kaks operatsiooni. Kõigepealt operatsiooniga *table(laskmine.seeria)* koostatakse vajalike veergudega tabel *seeria* ja siis tehakse operatsioon *select()*, mille sisendiks on esimese järgu tulemus ja mille väljundiks harjutused, millel on predikaadika määratud tüüp.

3. Kolmanda järgu tegevused (oranž) – operatsiooniga *join()*, mille sisendiks on teise järgu tulemused, seob kokku eelmises etapis leitud tabelid *seeria* ja *harjutus* kasutades selleks indeksit *laskmine.seeria.fk_seeria_harjutus_id*.

4. Neljanda järgu tegevused (lilla) – operatsiooniga *group by()* sooritatakse grupeerimis operatsioon eelnevas etapis loodud tabeli peal kasutades `GROUP BY` lauses määratud veergu.

5. Viienda e viimase järgu tegevused (punane) – operatsiooniga *project()* kuvatakse väljundis ainult `SELECT` lauses määratud veerud.

Täitmisplaani tehnilisema vaate nägemiseks tuleb kasutada võtmesõna `TRACE`. Selles tekstipõhises väljundis kuvatakse MonetDB andmebaasisüsteemis käivitatud MAL operatsioonid ja nendele kulunud aega mikrosekundites (*ticks*). Alljärgnev plaan on üksnes väljavõte reaalsest plaanist (natuke algust, natuke lõppu), sest need võivad olla võrdlemisi pikad, töös käsitletud lausete korral maksimaalselt 400 rida.

```
+--------+-------------------------------------------------------------------+
| ticks  | statement                                                         |
+========+===================================================================+
|      1 | X_3=0:int := sql.mvc();                                           |
|     26 | X_52=<tmp_1225>[0]:bat[:oid,:str] := bat.new(nil:oid,nil:str);    |

...

| 403182 | barrier X_329=false:bit{transparent} := language.dataflow();      |
|   1753 | sql.resultSet(X_66=<tmp_1230>[2]:bat[:oid,:str],X_68=<tmp_525>[2]: |
:        : bat[:oid,:str],X_70=<tmp_1225>[2]:bat[:oid,:str],X_72=<tmp_542>[2] :
:        : :bat[:oid,:int],X_74=<tmp_401>[2]:bat[:oid,:int],X_41=<tmp_1271>[1 :
:        : 000]:bat[:oid,:str],X_45=<tmp_1251>[1000]:bat[:oid,:sht]);        :
|      1 | end user.s4_1;                                                    |
| 406235 | function user.s4_1{autoCommit=true}(A0:str):void;                 |
| 406286 | X_4=0@0:void := user.s4_1("tyyp":str);                            |
+--------+-------------------------------------------------------------------+
```

Viimase lause `TRACE` andmed salvestatakse tabelisse *sys.tracelog*.

MonetDB andmebaasisüsteemis kuvati lause (`SELECT`, `UPDATE`, `INSERT`, `DELETE`) täitmisele kulunud aeg automaatselt tulemuse järel, peale selle lause käivitamist.

8. Testandmete genereerimine

Käesolevas töös testitakse andmebaasisüsteemide salvestusmeetodit, mis peaks kiirendama analüütilise iseloomuga päringuid. Teatavasti viiakse analüütilisi päringuid läbi suurte hulga andmete peal, seega pean ka ise parima tulemuse saavutamiseks genereerima võimalikult palju andmeid. Kahjuks ei saa antud töö raames rääkida gigabaitide või terabaitide suurustest andmemahtudest, kuna nende tekitamine oleks magistritöö mahtu arvestades ebapraktiline ja liiga aeganõudev. Siiski on vaja piisavalt andmeid, et töökiiruse erinevused tuleksid piisavalt hästi esile. Seda silmas pidades tegin andmete genereerimisel mõned lihtsustused ja kõrvalekaldumised reaalsest olukorrast, mis võiks esineda laskmise infosüsteemis. Täpsemalt genereeriti andmeid vastavalt järgnevatele eeldustele – võistlused toimuvad iga kahe nädala (14 päeva) tagant, võistlustest võtavad alati osa kõik laskjad ning iga laskja laseb alati kõiki võimalikke harjutusi.

Andmed on genereeritud selliselt, et need vastaks kõikidele andmebaasis kirjeldatud kitsendustele, sest muidu ei õnnestuks neid andmebaasi lisada. Testandmete väärtuste genereerimise loogika on sama, mis oli minu bakalaureusetöös (Puustusmaa, 2012).

Tabelitesse lisatakse järgmise suurusega andmeväärtused.

- Isiku eesnimi 3 kuni 10 tähte.
- Isiku perenimi 3 kuni 12 tähte.
- Isiku isikukood 11 kohaline arv.
- Klubi nimetus 5 kuni 12 tähte.
- Harjutus tüübi nimetus 10 kuni 25 tähte.
- Võistluse nimetus 8 kuni 55 tähte.
- Seeria summa 50 kuni 100 silma.

Ridade arvud tabelites on järgmised.

- Tabel *harjutus* 10 000 000 rida
- Tabel *harjutus_tyyp* 10 rida.
- Tabel *isik* 1010 rida.
- Tabel *klubi* 10 rida.

- Tabel *laskja* 1000 rida.
- Tabel *seeria* 60 000 000 rida.
- Tabel *voistlus* 1000 rida.

Kuna bakalaureusetöö jaoks oli testandmete generaator juba loodud ning andmebaasi struktuur on samasugune nagu bakalaureusetöös, siis ei oleks otstarbekas hakata turult uut generaatorit otsima või ise uut generaatorit kirjutada. Andmete genereerimiseks kasutatakse *Java* keeles kirjutatud programmi (vt lisa 5). Programmi põhiklassiks on `AndmeteGenereerija`, kus luuakse uus klassi `TestandmeteGeneraator` objekt ning kutsutakse selle peal välja `genereeri()` meetod (vt joonis 23). Genereeritud laused salvestatakse kõik eraldi failidesse. Faili nime struktuur on *laused_[tabeli_nimi].sql*, kus [tabeli_nimi] on asendatud reaalse tabeli nimega nagu *isik* e kokku tuleks siis *laused_isik.sql*.

```
public class AndmeteGenereerija {

    private static final String FAILI_EESLIIDE = "laused";

    public static void main(String[] args) {
        TestandmeteGeneraator testandmeteGeneraator = new TestandmeteGeneraator();
        testandmeteGeneraator.setBulkMode(true);
        testandmeteGeneraator.genereeri(FAILI_EESLIIDE);
        System.out.println("Valmis");
    }
}
```

Joonis 23. *Java* klass andmete genereerimiseks

Näiteks koodiosa, mis genereerib isikuid on järgmine (vt joonis 24).

```
// ISIK
System.out.println("Loon isikuid...");
List<String> isikukoodid = unikaalsedIsikukoodid(LASKJA + TREENER_JA_KLUBI);
for (String isikukood : isikukoodid) {
    String lause = looInsertLause("isik", new String[]{"isikukood", "eesnimi", "perenimi"}, isikukood,
            juhuslikTahtedeJada(EESNIMI_MIN_MAX_PIKKUS[0], EESNIMI_MIN_MAX_PIKKUS[1]),
            juhuslikTahtedeJada(PERENIMI_MIN_MAX_ARV_PIKKUS[0], PERENIMI_MIN_MAX_ARV_PIKKUS[1])
            );
    laused.add(lause);
}
kirjutaKoikFaili(laused, failiEesliide, "isik");
laused.clear();
```

Joonis 24. *Java* kood isikute loomiseks

Täiendavalt sooviks tähelepanu juhtida meetodi `setBulkMode()` (vt joonis 23) väljakutsele. Antud meetodi argument määrab ära, kas genereeritakse tavalised `INSERT` laused või laused hulga (*bulk*) stiilis. Hulga stiili korral on failis kirjas üksnes komadega eraldatud väärtused, mitte terviklik SQL `INSERT` lause. Failid, mis on loodud *bulk* stiilis on

keskmiselt viis korda väiksemad. Failide andmemahu võrdlused kõikide tabelite kohta on ära toodud alljärgnevas tabelis (vt tabel 3).

Tabel 3. `INSERT` lausete andmemahud erinevate stiilide korral

Faili nimi	**INSERT stiil, andmemaht megabaitides**	***Bulk* stiil, andmemaht megabaitides**	**Andmemahu erinevus kordades**
laused_harjutus.sql	1341	341	3,9
laused_harjutus_tyyp.sql	0,00072	0,00013	5,5
laused_isik.sql	0,08779	0,02783	3,2
laused_klubi.sql	0,00045	0,00007	6,4
laused_laskja.sql	0,12305	0,02949	4,2
laused_seeria.sql	4290	794	5,4
laused_voistlus.sql	0,11328	0,03418	3,3
KOKKU	5631,33	1135,09	5 (vahe 4496 MB)

Kuna nii Microsoft SQL Server andmebaasisüsteem kui ka MonetDB andmebaasisüsteem võimaldavad andmeid sisestada kasutades *bulk* meetodit, siis kasutatakse antud töös seda meetodit. Alljärgnevalt tuuakse näited mõlema andmebaasisüsteemi kohta.

Microsoft SQL Server

```
BULK INSERT isik
FROM 'C:\laused_isik.sql'
WITH (FIELDTERMINATOR = ',');
```

Kõik Microsoft SQL Server andmebaasisüsteemi jaoks mõeldud andmesisestuse laused on ära toodud lisades (vt lisa 6).

MonetDB

```
COPY INTO isik
FROM 'C:\laused_isik.sql'
USING DELIMITERS ',','\n';
```

Kõik MonetDB andmebaasisüsteemi jaoks mõeldud andmesisestuse laused on ära toodud lisades (vt lisa 7).

Kõlab soovitusi, et indeksid ja kitsendused tuleks lisada alles päras andmete sisestamist. Selline lähenemine peaks oluliselt vähendama andmete sisestamise aega, kuna ei teostata iga sisestatud rea korral kitsenduste kontrolli ega uuendata indekseid. Eelnevalt kirjeldatud andmemahtude juures olid andmete sisestamise ajad järgmised.

- Microsoft SQL Server – 15,5 minutit
- MonetDB – 1 minutit

Kitsenduste ja indeksite lisamine võttis aega järgmiselt.

- Microsoft SQL Server – 13 minutit
- Microsoft SQL Server (lisati ainult veerupõhine indeks) – 1 minut
- MonetDB – 9 minutit

Veendumaks, kas eelnev soovitus paika peab lisati prooviks ka andmed tabelitesse, kus kitsendused ja indeksid olid juba olemas.

- Microsoft SQL Server – 28 minutit
- Microsoft SQL Server (columnstore) – 3,5 minutit
- MonetDB – 14,3 minutit

Kuigi andmete sisestamise ajad olid kiiremad siis kui andmebaasis indekseid ja kitsendusi ei olnud, siis kumulatiivselt see erilist eelist ei andnud. Siiski, Microsoft SQL Serveri veerupõhise indeksiga andmebaasis toimus andmete lisamine oluliselt kiiremini kui veerupõhine indeks oli juba loodud. Seega veerupõhiseid indekseid kasutava Microsoft SQL andmebaasi puhul tuleb anda vastupidine soovitus – andmed tuleks lisada peale veerupõhiste indeksite loomist.

Peale andmete lisamist käivitasin Microsoft SQL Serveris järgmised statistika uuendamise laused.

- `UPDATE STATISTICS` *harjutus;*
- `UPDATE STATISTICS` *harjutus_tyyp;*
- `UPDATE STATISTICS` *isik;*
- `UPDATE STATISTICS` *klubi;*
- `UPDATE STATISTICS` *laskja;*
- `UPDATE STATISTICS` *seeria;*
- `UPDATE STATISTICS` *voistlus;*

MonetDB andmebaasides värskendasin statistikat järgmiste lausetega. (Table statistics, 2015)

- `ANALYZE laskmine.`*harjutus*;
- `ANALYZE laskmine.`*harjutus_tyyp*;
- `ANALYZE laskmine.`*isik*;
- `ANALYZE laskmine.`*klubi*;
- `ANALYZE laskmine.`*laskja*;
- `ANALYZE laskmine.`*seeria*;
- `ANALYZE laskmine.`*voistlus*;

9. Eksperimendi tulemused

Selles ja kõikides alljärgnevates peatükkides kasutatakse erinevatele andmebaasidele viitamiseks järgmiseid lühendeid.

- **MSR**: Microsoft SQL Serveri reapõhise salvestamisega andmebaas ilma veerupõhist indeksit kasutamata, kuid kasutades kõiki teisi indekseid ja kitsendusi.
- **MSV**: Microsoft SQL Serveri andmebaas kasutades veerupõhist indeksit, kuid kasutamata ühtegi teist täiendavat indeksit ja kitsendust.
- **MDBI**: MonetDB andmebaas kasutades indekseid ja kitsendusi.
- **MDB**: MonetDB andmebaas kasutamata ühtegi täiendavat indeksit ja kitsendust.

Esiteks tuuakse välja andmemahtude erinevused andmebaaside vahel. Tabelis 4 tuuakse välja andmebaaside andmemahud kettal, enne ja pärast indeksite ning kitsenduste lisamist. Parimad tulemused on ära märgitud tumedalt.

Tabel 4. Andmebaaside andmemahud

Andmebaas	**Enne indeksite ja võtmete lisamist megabaitides**	**Pärast indeksite ja võtmete lisamist megabaitides**
MSR	1380	2130
MSV	1380	**211**
MDBI	**670**	2040
MDB	**670**	670 (ühtegi täiendavat indeksit või võtit ei lisatud)

Andmebaasisüsteemi Microsoft SQL Server puhul kasutasin andmebaasi andmemahu mõõtmiseks Microsoft Server 2014 Management Studio poolt pakutavat standardset raportit (*database > reports > Disk Usage* ning sealt *Disk Space Used By Data Files > Space Used*).

Andmebaasisüsteemi MonetDB puhul kasutasin andmebaasi andmemahu mõõtmiseks andmebaasi failide kausta suuruse mõõtmist. Antud kausta asukoht määratakse ära MonetDB installatsiooni kaustas olevas failis *M5Server.bat* `MONETDBFARM` nimelise muutujaga.

Teiseks koondatakse kõikide päringute tulemused operatsioonide kaupa alljärgnevasse tabelisse (vt tabel 5). Tumedalt on ära toodud kiireim operatsiooni aeg üle kõikide andmebaaside ning käivitamise viiside (külm/soe).

Tabel 5. Testide tulemused operatsioonide kaupa

Test	**MSR** ***Külm***	**MSR** ***Soe***	**MSV** ***Külm***	**MSV** ***Soe***	**MDBI** ***Külm***	**MDBI** ***Soe***	**MDB** ***Külm***	**MDB** ***Soe***
Päring 1	36s	5,0s	6,5s	1,0s	128s	26s	90s	**0,4s**
Päring 2	37s	5,4s	6,6s	**1,2s**	146s	27s	141s	**1,2s**
Päring 3	10s	0,3s	0,9s	**0,04s**	15s	0,2s	1,5s	0,1s
Päring 4	7s	**0,01s**	6s	0,6s	115s	23s	52s	0,03s
Päring 5	0,2s	**0,001s**	0,6s	0,01s	1s	0,02s	0,7s	0,02s
Lisamine	0,1s	**0,001s**	0,2s	**0,001s**	60s	14s	0,2s	0,05s
Uuendamine	0,1s	**0,03s**	0,3s	**0,03s**	1,4s	0,1s	0,9s	0,08s
Kustutamine	0,1s	**0,05s**	0,2s	**0,05s**	24s	61s	23s	57s

Külm – Enne lausete käivitamist tühjendasin andmebaasisüsteemide poolt kasutatava puhvri. Tehtud tegevused on detailsemalt kirjeldatud seitsmendas peatükis.

Soe – Tähendab, et enne lause käivitamist on sama lause juba ükskord käivitatud ja andmebaasisüsteemi puhvrit pole vahepeal tühjaks tehtud.

Alljärgnevas tabelis (vt tabel 6) on ära toodud samad tulemused, mis tabelis 5, kuid grupeeritult andmebaasisüsteemide kaupa. Tumedalt on ära toodud kiireim operatsiooni aeg vastavas andmebaasisüsteemis (Microsoft SQL Server või MonetDB).

Tabel 6. Testide tulemused andmebaasisüsteemide kaupa

Test	**MSR** ***Külm***	**MSR** ***Soe***	**MSV** ***Külm***	**MSV** ***Soe***	**MDBI** ***Külm***	**MDBI** ***Soe***	**MDB** ***Külm***	**MDB** ***Soe***
Päring 1	36s	5,0s	6,5s	**1,0s**	128s	26s	90s	**0,4s**
Päring 2	37s	5,4s	6,6s	**1,2s**	146s	27s	141s	**1,2s**
Päring 3	10s	0,3s	0,9s	**0,04s**	15s	0,2s	1,5s	**0,1s**
Päring 4	7s	**0,01s**	6s	0,6s	115s	23s	52s	**0,03s**
Päring 5	0,2s	**0,001s**	0,6s	0,01s	1s	**0,02s**	0,7s	**0,02s**
Lisamine	0,1s	**0,001s**	0,2s	**0,001s**	60s	14s	0,2s	**0,05s**
Uuendamine	0,1s	**0,03s**	0,3s	**0,03s**	1,4s	0,1s	0,9s	**0,08s**
Kustutamine	0,1s	**0,05s**	0,2s	**0,05s**	24s	61s	**23s**	57s

10. Eksperimendi tulemuste analüüs

Selles peatükis vaadatakse saadud tulemuste taha. Andmekäitluse operatsioonide korral tähendab see ka lausete täitmisplaanide uurimist. Parema loetavuse huvides korratakse alapeatükkides infot, mis on juba välja toodud koondtabelites (vt tabel 5 ja tabel 6).

10.1 Andmebaaside andmemahtude analüüs

Mitmekordsed andmebaaside andmemahtude erinevused tulenesid sellest, et veerupõhise salvestamise korral andmed ka pakitakse. Parimat tulemust näitas MSV, mille korral andmebaasi suurus oli vaid 211 MB. Andmebaasi MDB andmemaht oli sellest juba üle kolme korra suurem. MSR ja MSV andmemahtude vahe oli täpselt seitse korda, mis läheb kokku sellega, mida Microsoft oma veebilehel ka lubas. „Use the columnstore index to achieve up to 10x query performance gains over traditional row-oriented storage, and up to 7x data compression over the uncompressed data size.“ (Columnstore Indexes Described, 2015)

SQL-andmebaasides moodustavad üldjuhul suurema osa andmemahust baastabelites olevad andmed ja nendele loodud indeksid. Näiteks MSR puhul moodustasid indeksid 35% andmemahust aga MDBI korral moodustasid indeksid koguni 64% andmemahust. Reapõhistes andmebaasisüsteemides kasutatakse indekseid selleks, et parandada päringute kiiruseid, sest indeksite puudumisel tuleks alati läbi vaadata kõik tabeli read. See aga on aeglane operatsioon. Samas, sõltuvalt päringust ja andmemahust võib see ka olla kõige optimaalsem operatsioon. Veerupõhiste andmebaasisüsteemide korral ei pruugi kõikide ridade läbivaatamine enam väga kulukas olla, sest veergude andmed on kompaktselt koos ja seega on neid võimalik kiiremini järjest töödelda. Siit järeldub, et olenevalt andmebaasi disainist ja seal läbiviidavatest andmetöötluse operatsioonidest, ei pruugi veerupõhise salvestusega andmebaasisüsteemides enam olla vajadust indekseid luua. Minu katse MonetDB andmebaasisüsteemiga näitas, et indeksite loomine andmebaasis (MDBI) ei parandanud ühegi operatsiooni kiirust, aga samas suurendas oluliselt andmebaasi andmemahtu.

10.2 Andmekäitluse operatsioonide täitmise kiiruse analüüs

Lausete täitmise kiiruse analüüsis toon kiiruste erinevused välja kordades, sest see ilmestab aegade erinevust paremini olukorras, kus kahe lause täitmise ajaline vahe on väike. Samas tuleb meeles pidada, et kui näiteks kahe lause täitmise kiirus erineb mitukümmend korda, siis see võib tähendada nii paarikümne millisekundilist kui ka paarikümne sekundilist erinevust lause täitmise ajas.

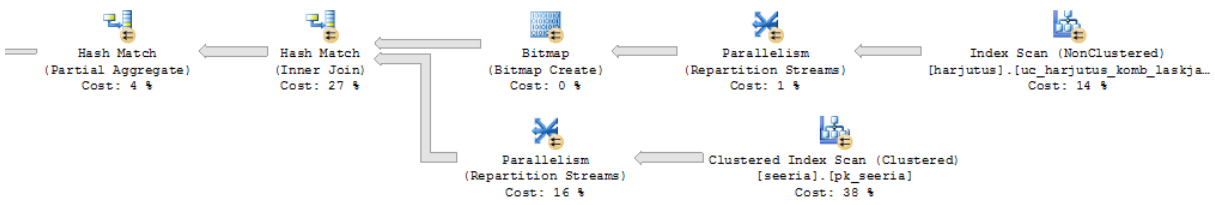
Kui kiiruste vahed olid liiga väikesed, et väljendada seda kordades, kasutasin protsentuaalset võrdlemist.

10.2.1 Päring 1

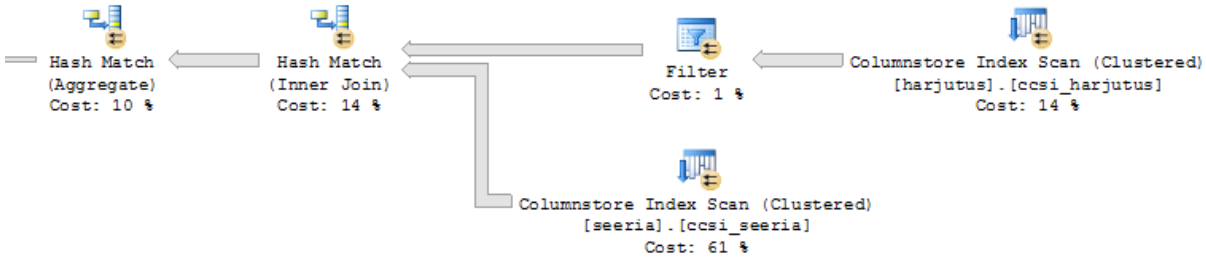
Laskuri parim seeria harjutuse tüübis *HT*.
Väljund: harjutus.isikukood_isik_laskja, max(summa) AS „parim seeria“
Predikaat: harjutus.nimetus_harjutus_tyyp = *HT*

Test	**MSR** ***Külm***	**MSR** ***Soe***	**MSV** ***Külm***	**MSV** ***Soe***	**MDBI** ***Külm***	**MDBI** ***Soe***	**MDB** ***Külm***	**MDB** ***Soe***
Päring 1	36s	5,0s	6,5s	1,0s	128s	26s	90s	**0,4s**

MSR vs. MSV
Võrreldes päringu täitmisplaane (vt joonis 25 ja joonis 26) näeme, et mõlemas andmebaasis läheb kõige rohkem aega seeriate ülesotsimisele. Seeriate leidmiseks vaadatakse järjest läbi (*scan* operatsioon) indeks `pk_seeria` või `ccsi_seeria`. Esimese puhul on tegemist reapõhise salvestamisega andmebaasis primaarvõtmele automaatselt loodud klasterdatud indeksiga, teisel juhul tabelile *seeria* loodud klasterdatud veerupõhise indeksiga. MSR andmebaasis on operatsioon aeglasem, sest tabeli *seeria* jaoks on vaja lugeda üle 30000 lehekülje rohkem, samas tabeli *harjutus* suhte kehtib vastupidine olukord (vt tabel 7 – *logical reads*).



Joonis 25. Päring 1 täitmisplaan MSR korral



Joonis 26. Päring 1 täitmisplaan MSV korral

Tabel 7. Päring 1 IO statistika MSR ja MSV korral

Reapõhise salvestusega andmebaasi `IO` statistiline väljund
`Table 'harjutus'. Scan count 9, logical reads 53939, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.` `Table 'seeria'. Scan count 9, logical reads 127292, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.`
Veerupõhise salvestusega andmebaasi `IO` statistiline väljund
`Table 'harjutus'. Scan count 8, logical reads 87100, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.` `Table 'seeria'. Scan count 8, logical reads 91744, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.`

MDBI vs. MDB

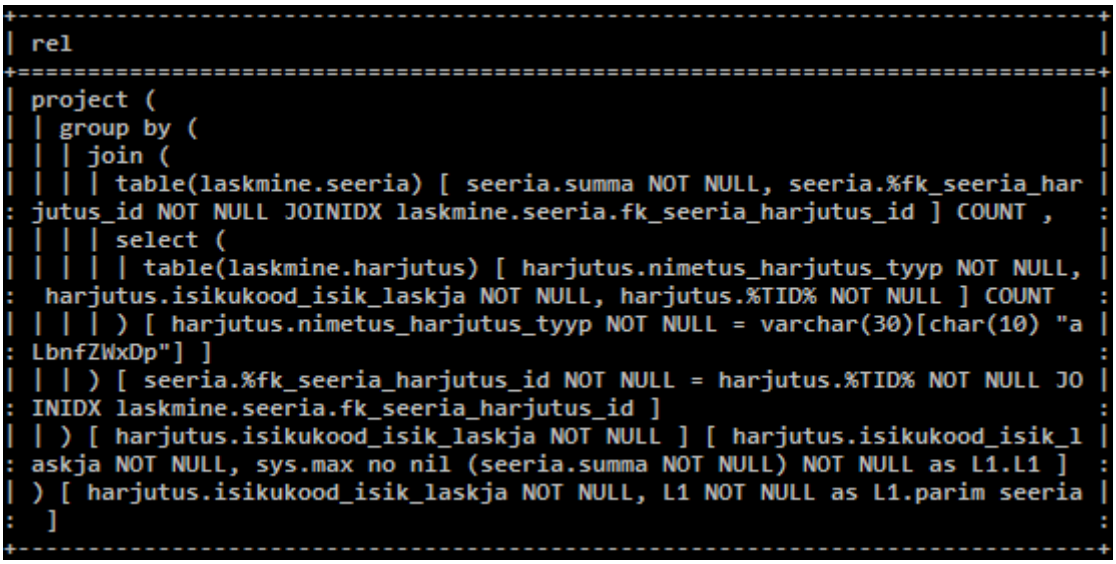
Indeksitega MonetDB andmebaasis oli päring 1 üllatuslikult väga aeglane. Vaadates indeksitega (MDBI) andmebaasis toimunud päringu täitmisplaani (vt joonis 28) näeme, et tabeliga *seeria* ühendatakse harjutused läbi välisvõtme indeksi (`JOINIDX`) `fk_seeria_harjutus_id`, kuid MDB korral (vt joonis 27) see nii ei ole. Nii suure hulga ridade (10% tabelist *seeria*) ühendamine läbi indeksi aga osutub MonetDB andmebaasisüsteemis kulukaks. Vaadates käsuga `TRACE` reaalselt rakendatud täitmisplaani näemegi, et just palju aega läheb funktsiooni `algebra.join()` väljakutsetele.

Alljärgnevalt on ära toodud väljavõte sellest täitmisplaanist, kus antud funktsiooni väljakutse on märgitud tumedalt.

```
+-----------+-------------------------------------------------------------------+
| ticks     | stmt                                                              |
+===========+===================================================================+
|         2 | X_3 := sql.mvc();                                                 |
|        32 | X_77:bat[:oid,:oid] =<tmp_25>[7500000] := sql.tid(X_3=0,"laskmi   |
:           : ne","seeria",0,8);                                                :
|        12 | X_13:bat[:oid,:str] =<tmp_610>[10000000] := sql.bind(X_3=0,"las   |
:           : kmine","harjutus","nimetus_harjutus_tyyp",0);                     :
|         7 | X_36:bat[:oid,:sht] =<tmp_767>[0] := sql.bind(X_3=0,"laskmine",   |
:           : "seeria","summa",1);                                              :
...
|        10 | (X_108:bat[:oid,:oid] =<tmp_773>[0],X_109:bat[:oid,:oid] =<tmp_   |
:           : 773>[0]) := sql.bind_idxbat(X_3=0,"laskmine","seeria","fk_seeri   :
:           : a_harjutus_id",2,4,8);                                            :
|         2 | X_96:bat[:oid,:oid] =<tmp_1246>[7500000] := sql.bind_idxbat(X_3   |
:           : =0,"laskmine","seeria","fk_seeria_harjutus_id",0,4,8);            :
|         2 | X_85:bat[:oid,:oid] =<tmp_676>[7500000] := sql.tid(X_3=0,"laskm   |
:           : ine","seeria",4,8);                                               :
...
|         7 | X_158=<tmp_1247>[7500000] := sql.projectdelta(X_85=<tmp_676>:ba   |
:           : t[:oid,:oid][7500000],X_96=<tmp_1246>:bat[:oid,:oid][7500000],X   :
:           : _108=<tmp_773>:bat[:oid,:oid][0],X_109=<tmp_773>:bat[:oid,:oid]   :
:           : [0]);                                                             :
...
|         2 | X_20=<tmp_1262>[1000000] := X_18=<tmp_1262>[1000000];             |
...
| 148849458 | (X_173=<tmp_12>[750000],X_174=<tmp_741>[750000]) := algebra.joi   |
:           : n(X_158=<tmp_1247>[7500000],X_20=<tmp_1262>[1000000]);            :
...
|         1 | end s2_1;                                                         |
| 149677949 | function user.s2_1(A0="aLbnfZWxDp");                              |
| 149678003 | X_5:void  := user.s2_1("aLbnfZWxDp");                             |
+-----------+-------------------------------------------------------------------+
```

```
+-----------------------------------------------------------------------------------+
| rel                                                                               |
+===================================================================================+
| project (                                                                         |
| | group by (                                                                      |
| | | join (                                                                        |
| | | | select (                                                                    |
| | | | | table(laskmine.harjutus) [ harjutus.harjutus_id NOT NULL, harjutus.       |
: nimetus_harjutus_tyyp NOT NULL, harjutus.isikukood_isik_laskja NOT NULL ] C       :
: OUNT                                                                              :
| | | | ) [ harjutus.nimetus_harjutus_tyyp NOT NULL = varchar(30)[char(10) "a     |
: LbnfZWxDp"] ],                                                                    :
| | | | table(laskmine.seeria) [ seeria.harjutus_id NOT NULL, seeria.summa NO     |
: T NULL ] COUNT                                                                    :
| | | ) [ harjutus.harjutus_id NOT NULL = seeria.harjutus_id NOT NULL ]           |
| | ) [ harjutus.isikukood_isik_laskja NOT NULL ] [ harjutus.isikukood_isik_l     |
: askja NOT NULL, sys.max no nil (seeria.summa NOT NULL) NOT NULL as L1.L1 ]        :
| ) [ harjutus.isikukood_isik_laskja NOT NULL, L1 NOT NULL as L1.parim seeria     |
:  ]                                                                               :
+-----------------------------------------------------------------------------------+
```

Joonis 27. Päring 1 täitmisplaan MDB korral



Joonis 28. Päring 1 täitmisplaan MDBI korral

MSV vs. MDB

Päring 1 oli ilma indeksiteta MonetDB andmebaasis ligi kaks korda kiirem kui veerupõhise indeksiga Microsoft SQL Server andmebaasis.

10.2.2 Päring 2

Laskuri parim seeria harjutuse tüübis HT.

Väljundi veerud: harjutus.isikukood_isik_laskja, MAX(summa)

Predikaat: harjutus.harjutus_id != 0 AND harjutus.voistlus_id != 0 AND harjutus.nimetus_harjutus_tyyp = HT AND harjutus.isikukood_isik_laskja != '' AND seeria.harjutus_id != 0 AND seeria_number != 0 AND seeria.summa != 0

Test	**MSR** ***Külm***	**MSR** ***Soe***	**MSV** ***Külm***	**MSV** ***Soe***	**MDBI** ***Külm***	**MDBI** ***Soe***	**MDB** ***Külm***	**MDB** ***Soe***
Päring 2	37s	5,4s	6,6s	**1,2s**	146s	27s	141s	**1,2s**

MSR vs. MSV

Reapõhises andmebaasis läks päring 2 8% aeglasemaks ja veerupõhises andmebaasis 20% aeglasemaks võrreldes päringuga 1. Vajaminevate lehekülgede arv jäi reapõhise andmebaasi jaoks enamvähem samaks, kuid veerupõhise andmebaasi jaoks mitte (vt tabel 8). Veerupõhine andmebaas pidi tabeli *seeria* jaoks lugema 90 000 lehekülge rohkem ja tabeli *harjutus* jaoks 600 lehekülge rohkem. See tuleneb sellest, et veerupõhise salvestamise korral on erinevate

veergude väärtused salvestatud erinevatel lehekülgedel ja iga uue veeru lisamine predikaati põhjustab vähemalt ühe täiendava lehekülje lugemise.

Tabel 8. Päring 2 IO statistika MSR ja MSV korral

Reapõhise salvestusega andmebaasi `IO` statistiline väljund
`Table 'harjutus'. Scan count 9, **logical reads 53654**, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.` `Table 'seeria'. Scan count 9, **logical reads 127272**, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.`
Veerupõhise salvestusega andmebaasi `IO` statistiline väljund
`Table 'harjutus'. Scan count 8, **logical reads 87696**, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.` `Table 'seeria'. Scan count 8, **logical reads 183488**, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.`

MDBI vs. MDB

Võrreldes päringuga 1 läks päring 2 MDBI andmebaasis natukene aeglasemaks, kuid MDB andmebaasis ligi kolm korda aeglasemaks. Päring 2 oli kokkuvõttes endiselt indeksitega MonetDB andmebaasis oluliselt aeglasem kui indeksiteta MonetDB andmebaasis. Põhjus on sama, mis päring 1 korral.

MSV vs. MDB

MSV andmebaasis muutus päring 2 võrreldes päringuga 1 20% aeglasemaks, MDB andmebaasis muutus päring 2 juba 200% (0,8 sekundit) aeglasemaks kui päring 1 samas andmebaasis. Teisisõnu, veergude lisamine päringusse (predikaati) mõjutas rohkem MonetDB andmebaasisüsteemi kui Microsoft SQL Server andmebaasisüsteemi.

10.2.3 Päring 3

Aastal *A* võistlustest osa võtnud laskurite arv.

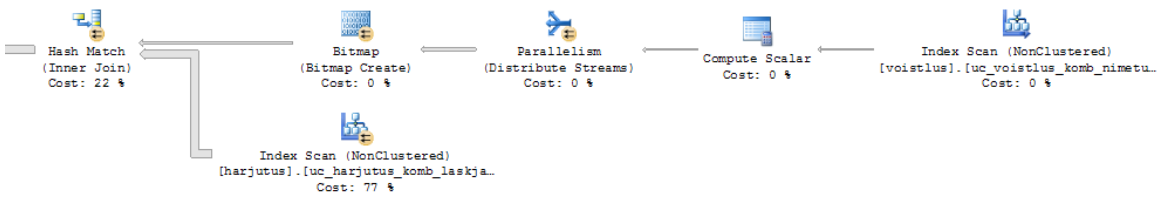
Väljund: YEAR(voistlus.alguse_kuupaev) AS aasta, COUNT(DISTINCT harjutus.isikukood_isik_laskja) AS laskjaid

Predikaat: YEAR(voistlus.alguse_kuupaev) = *A*

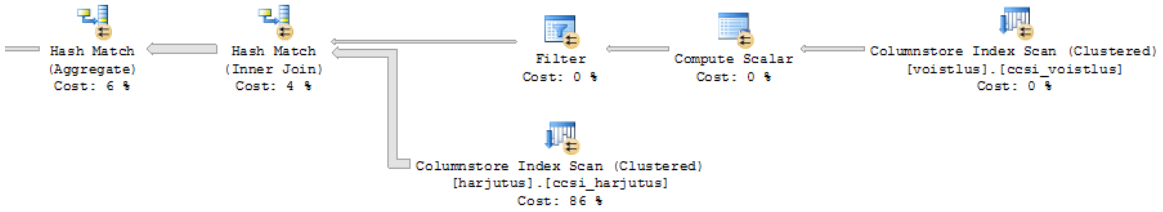
Test	MSR *Külm*	MSR *Soe*	MSV *Külm*	MSV *Soe*	MDBI *Külm*	MDBI *Soe*	MDB *Külm*	MDB *Soe*
Päring 3	10s	0,3s	0,9s	**0,04s**	15s	0,2s	1.5s	0,1s

MSR vs. MSV

Päring 3 oli MSV andmebaasis ligi seitse ja pool korda kiirem kui MSR andmebaasis. Mõlemas andmebaasis läks põhiline aeg harjutuste üles otsimiseks, et neid siduda vastava võistlusega (vt joonis 29 ja 30). Reapõhises andmebaasis kasutatakse harjutuste ülesleidmiseks unikaalsuse kitsenduse jõustamiseks loodud indeksit `uc_harjutus_komb_laskja_ja_voistlus_ja_harjutus_tyyp` (vt joonis 29) ja veerupõhises andmebaasis indeksit `ccsi_harjutus` (vt joonis 30). Unikaalsuse kitsenduse indeksi probleemiks on ainult veergude mitte-optimaalne järjestus. Antud indeksi võtmes on esimeseks veeruks `isikukood_isik_laskja` mitte `voistlus_id`. See tähendab, et indeksi võti on primaarselt sorteeritud isikukoodi järgi ja alles siis teiste veergude järgi. See omakorda toob kaasa antud päringu jaoks üleliigsete indeksi lehekülgede lugemise. Reapõhises andmebaasis oli vaja tabeli *harjutus* jaoks lugeda üle 50000 lehekülje aga veerupõhises andmebaasis ligi 30000 lehekülge (vt tabel 9).



Joonis 29. Päring 3 täitmisplaan MSR korral



Joonis 30. Päring 3 täitmisplaan MSV korral

Tabel 9. Päring 3 IO statistika MSR ja MSV korral

Reapõhise salvestusega andmebaasi `IO` statistiline väljund

`Table 'harjutus'. Scan count 9, **logical reads 53939**, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.`
 	
`Table 'voistlus'. Scan count 1, **logical reads 6**, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.`

Veerupõhise salvestusega andmebaasi `IO` statistiline väljund

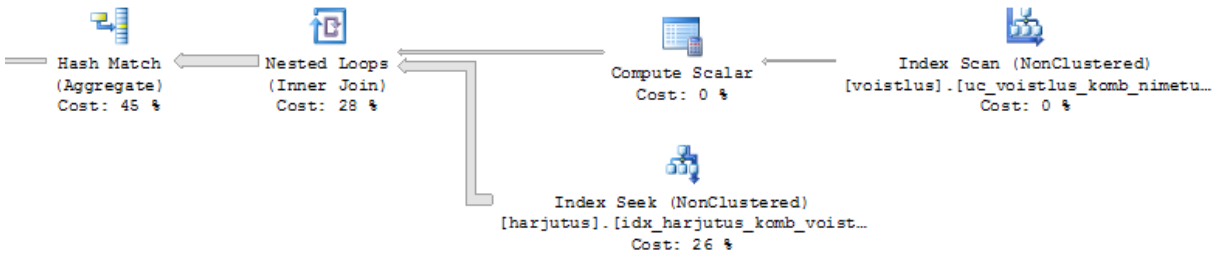
`Table 'harjutus'. Scan count 8, **logical reads 29740**, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.`
 	
`Table 'voistlus'. Scan count 8, **logical reads 0**, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.`

Leidmaks, kas sobilikuma veergude järjekorraga indeks vähendaks vajaminevate lehekülgede arvu ja parandaks päringu kiirust, lõin MSR andmebaasis ajutiselt ühe täiendava indeksi.

```
CREATE INDEX idx_harjutus_komb_voistlus_id_ja_isikukood_isik_laskja ON
harjutus (voistlus_id, isikukood_isik_laskja);
```

Nüüd otsustas Microsoft SQL Server andmebaasisüsteem kasutada päringu 3 täitmiseks seda indeksit (vt joonis 31). Sellega seoses vähenes vajaminevate lehekülgede arv ligi 50 korda (vt tabel 10). Seda tänu sellele, et andmebaasisüsteem saab kasutada uue indeksi peal indeksi põhist otsingu operatsiooni (*index seek* operatsioon) e otsitakse igat harjutust eraldi (vt tabel 10 – *scan count*). Päringu töötlemise aeg külmalt vähenes kümme korda (vähem lehekülgi oli vaja kettalt sisse lugeda), soojalt jäi see siiski samaks, kuigi harjutuste otsimise kulu vähenes 77 protsendilt 26 protsendile. Põhjus seisneb selles, et kuigi andmelehekülgi oli vaja lugeda

oluliselt vähem, jäi endiselt alles andmete töötlemise ajaline kulu. Seega antud päringu täitmise juures on kulukam osa pigem andmete töötlemine, kui vajalike andmete lugemine kettalt või muutmälust.



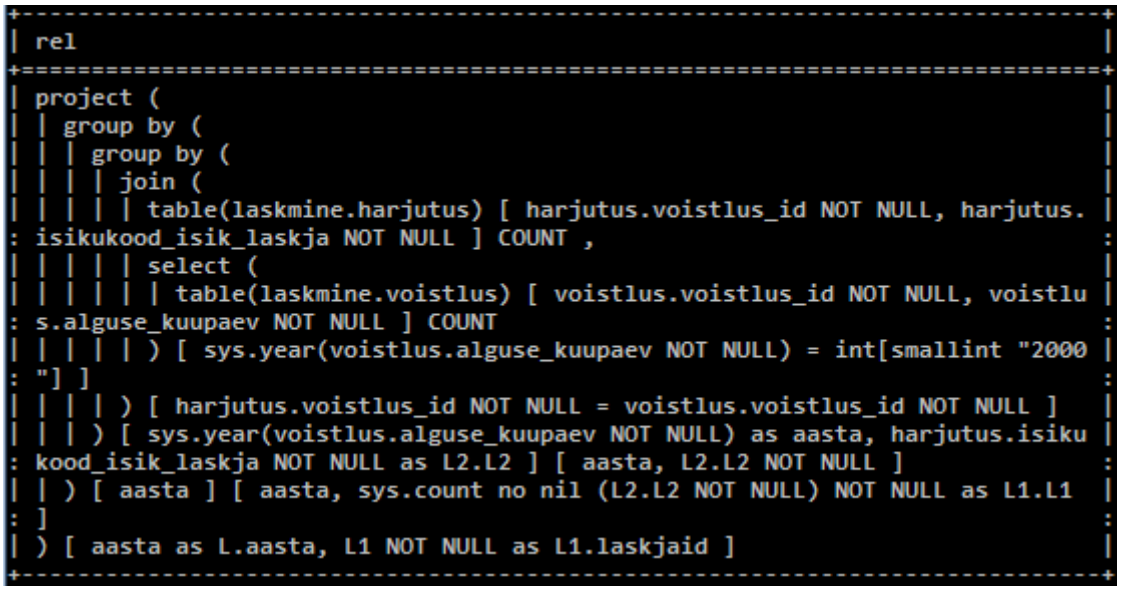
Joonis 31. Päring 3 täitmisplaan Microsoft SQL Serveris täiendava indeksiga (reapõhine)

Tabel 10. Päring 3 IO statistika MSR korral kasutades täiendavat indeksit

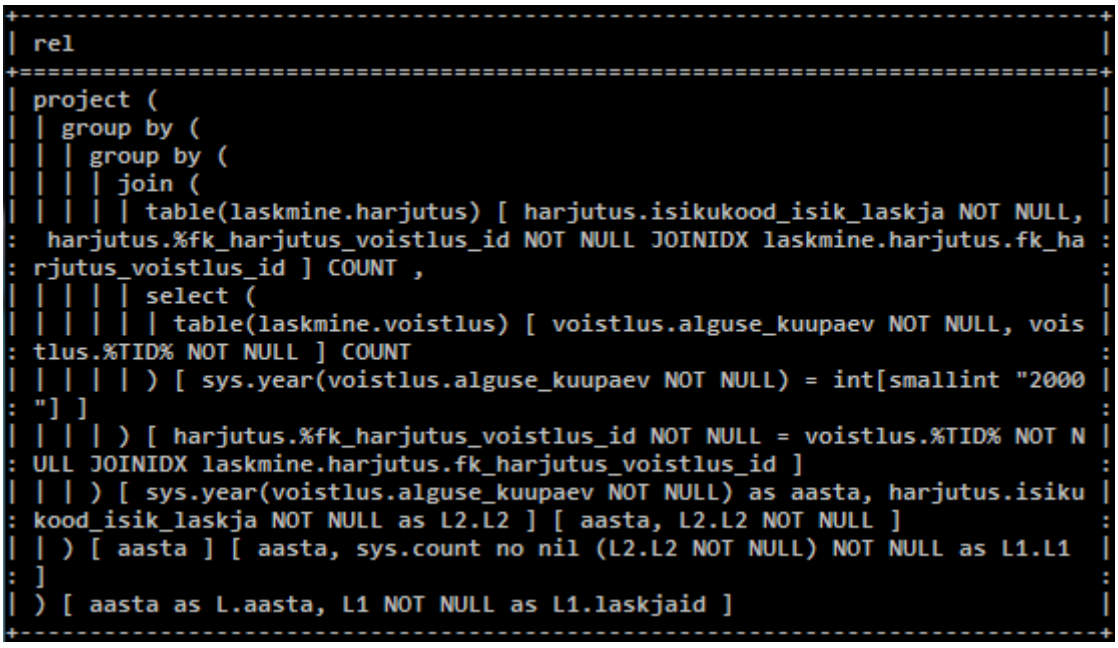
Reapõhise salvestusega andmebaasi `IO` statistiline väljund
`Table 'harjutus'. Scan count 27, logical reads 1105, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.`

MDBI vs. MDB

Päring 3 oli külmalt MDBI andmebaasis 10 korda aeglasem, kui MDB andmebaasis. Soojalt toimus aga päringu 3 täitmine mõlemas andmebaasis sama kiirelt. Päringu 3 täitmine erineb nendes kahes andmebaasis selle poolest, et MDBI andmebaasis kasutatakse tabelite *harjutus* ja *voistlus* ühendamiseks indeksit (`JOINIDX`) `fk_harjutus_voistlus_id` (vt joonis 33), kuid andmebaasis MDB seda ei tehta (vt joonis 32). See tähendab, et MDBI andmebaasi korral peab andmebaasisüsteem (esimene e külm päringu käivitus) ka indeksid kettalt mällu lugema. Seekord ei põhjusta indeksi kasutamine tabelite ühendamiseks suurt päringu kiiruse vähenemist, nagu seda oli eelnevate päringute puhul, sest predikaadile vastab võrdlemisi vähe ridu.



Joonis 32. Päring 3 täitmisplaan MDB korral



Joonis 33. Päring 3 täitmisplaan MDBI korral

MSR vs. MDBI

Päring 3 töötas reapõhises Microsoft SQL Server andmebaasis enam-vähem sama kiirelt kui indeksitega MonetDB andmebaasis.

MSV vs. MDB

Päring 3 oli ilma indeksiteta MonetDB andmebaasis ligi kaks ja pool korda aeglasem kui veerupõhise indeksiga Microsoft SQL Server andmebaasis.

10.2.4 Päring 4

Laskuri *L* rekord harjutuse tüübis *HT*.

Väljund: MAX(tulemused.tulemus) AS ’parim tulemus’ (harjutus.isikukood_isik_laskja, SUM(seeria.summa) AS tulemus)

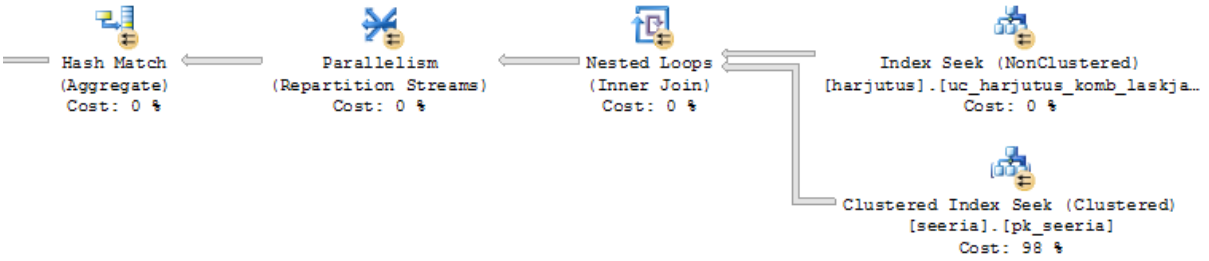
Predikaat: harjutus.nimetus_harjutus_tyyp = *HT* AND harjutus.isikukood_isik_laskja = *L*

Test	**MSR** ***Külm***	**MSR** ***Soe***	**MSV** ***Külm***	**MSV** ***Soe***	**MDBI** ***Külm***	**MDBI** ***Soe***	**MDB** ***Külm***	**MDB** ***Soe***
Päring 4	7s	**0,01s**	6s	0,6s	115s	23s	52s	0,03s

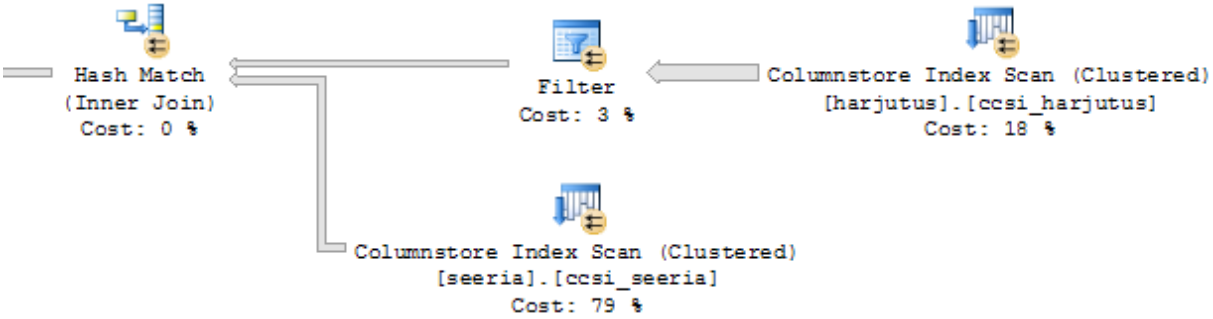
MSR vs. MSV

Antud päring oli reapõhises andmebaasis juba 60 korda kiirem kui veerupõhises andmebaasis. Võrreldes täitmisplaane (vt joonis 34 ja 35) näeme, et selle päringu juures otsustas andmebaasisüsteem, reapõhise andmebaasi puhul, kasutada indeksi põhist otsingut (*index seek* operatsiooni). Reapõhises Microsoft SQL Server andmebaasis kasutatakse kitsenduse `uc_harjutus_komb_laskja_ja_voistlus_ja_harjutus_tyyp` indeksit, et üles leida konkreetsed harjutused ja tabeli *seeria* klasterdatud primaarvõtme indeksit, et üles leida konkreetsed seeriad.

Kuna veerupõhine indeks salvestab andmeid teistmoodi kui tavaline klasterdatud või mitte-klasterdatud indeks, siis ei ole neis võimalik läbi viia indeksi põhist otsingut e *index seek* tüüpi operatsiooni. Teisisõnu, veerupõhise salvestamisega andmebaas jääb alati kasutama indeksipõhist tabeli läbivaatamist e *index scan* operatsiooni. Selletõttu erinevad ka pärigu jaoks vajaminevad lehekülgede arvud nii kardinaalselt. Reapõhise andmebaasi korral on päringu täitmiseks vaja kokku lugeda natuke üle 3000 lehekülje aga veerupõhise andmebaasi korral juba ligi 180000 lehekülge (vt tabel 11).



Joonis 34. Päring 4 täitmisplaan MSR korral



Joonis 35. Päring 4 täitmisplaan MSV korral

Tabel 11. Päring 4 IO statistika MSR ja MSV korral

Reapõhise salvestusega andmebaasi `IO` statistiline väljund

`Table 'seeria'. Scan count 1000, **logical reads 3257**, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.`
 	
`Table 'harjutus'. Scan count 9, **logical reads 62**, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.`

Veerupõhise salvestusega andmebaasi `IO` statistiline väljund

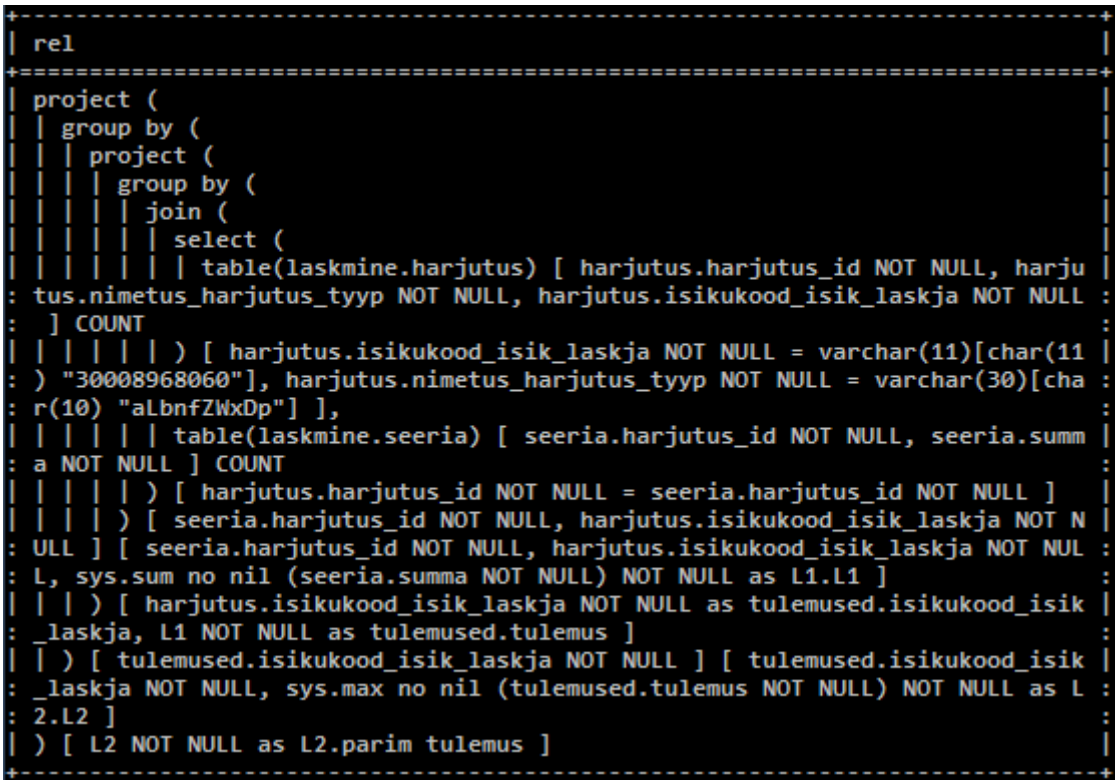
`Table 'seeria'. Scan count 8, **logical reads 91744**, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.`
 	
`Table 'harjutus'. Scan count 8, **logical reads 87100**, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.`

MDBI vs. MDB

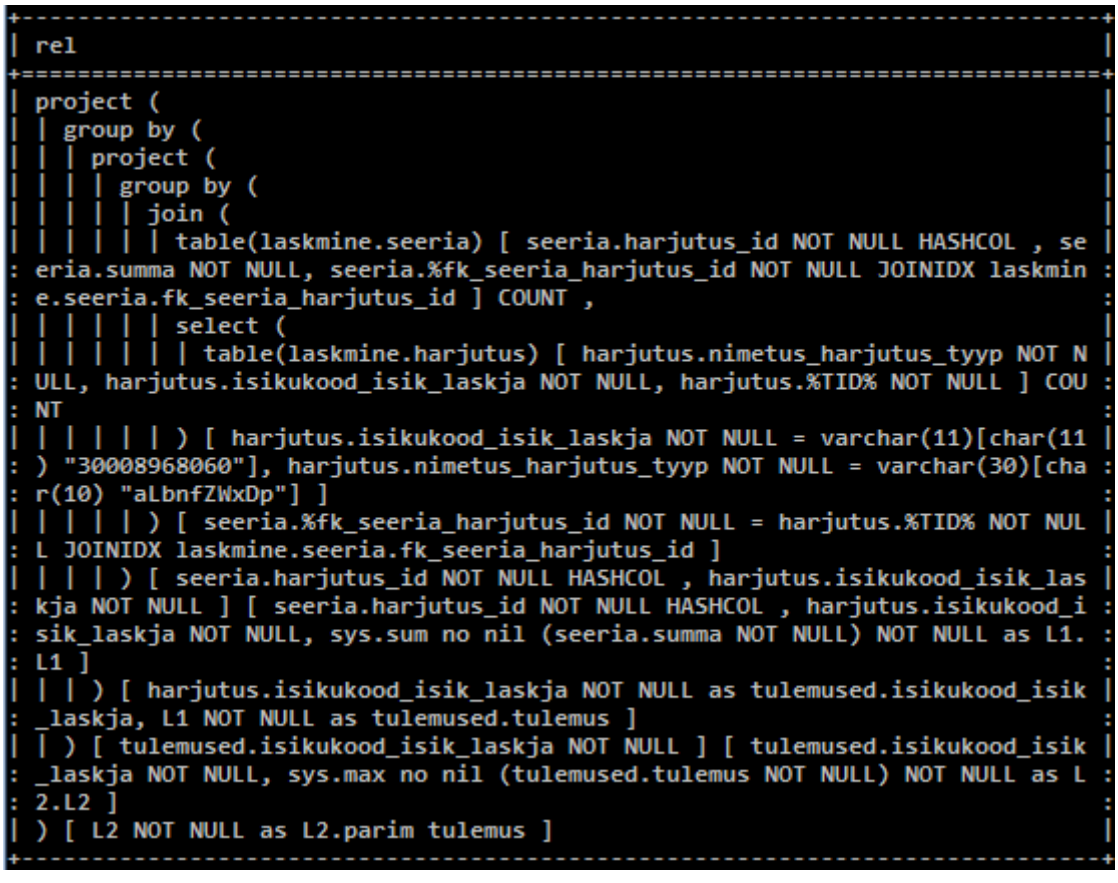
Indeksitega MonetDB andmebaasis oli päring 4 oluliselt (umbes 766 korda) aeglasem kui indeksiteta MonetDB andmebaasis. See on võrdlemisi üllatav, kuna antud päringu juures võiks indeksitest kasu olla. Tuleb välja, et MDBI andmebaasis ei kasutata harjutuste üles otsimiseks unikaalsuse kitsenduse indeksit nagu seda oli MSR andmebaasi korral, siiski ei saa see olla aegluse põhjuseks, sest MDB andmebaasis pole ühtegi täiendavat indeksit. Vaadates päringu reaalset täitmisplaani (`TRACE`) näeme, et kõige rohkem aega võtab jällegi `algebra.join()` funktsioon. Alljärgnevas täitmisplaani väljavõttes on ära toodud üks mitmest (märgitud tumedalt) antud funktsiooni väljakutsest.

```
+----------+--------------------------------------------------------------------+
| ticks    | stmt                                                               |
+==========+====================================================================+
|        1 | X_4 := sql.mvc();                                                  |
...
|        9 | (X_124:bat[:oid,:oid] =<tmp_773>[0],X_125:bat[:oid,:oid] =<tmp_7 |
:          : 73>[0]) := sql.bind_idxbat(X_4=0,"laskmine","seeria","fk_seeria_ :
:          : harjutus_id",2,5,8);                                              :
|        3 | X_110:bat[:oid,:oid] =<tmp_44>[7500000] := sql.bind_idxbat(X_4=0 |
:          : ,"laskmine","seeria","fk_seeria_harjutus_id",0,5,8);              :
|        3 | X_100:bat[:oid,:oid] =<tmp_1251>[7500000] := sql.tid(X_4=0,"lask |
:          : mine","seeria",5,8);                                              :
...
|        4 | X_206=<tmp_1250>[7500000] := sql.projectdelta(X_100=<tmp_1251>:b |
:          : at[:oid,:oid][7500000],X_110=<tmp_44>:bat[:oid,:oid][7500000],X_ :
:          : 124=<tmp_773>:bat[:oid,:oid][0],X_125=<tmp_773>:bat[:oid,:oid][0 :
:          : ]);                                                               :
...
| 23796919 | (X_221=<tmp_651>[750],X_222=<tmp_1264>[750]) := algebra.join(X_2 |
:          : 06=<tmp_1250>[7500000],X_28=<tmp_1274>[1000]);                    :
...
|        1 | end s3_1;                                                          |
| 78559215 | function user.s3_1(A0="aLbnfZWxDp",A1="30008968060");              |
| 78559276 | X_5:void  := user.s3_1("aLbnfZWxDp","30008968060");                |
+----------+--------------------------------------------------------------------+
```

See tähendab, et ka selle päringu puhul on probleemiks indeksi põhine *join* operatsioon, täpsemalt seeriate ühendamine harjutustega läbi indeksi (`JOINIDX`) `fk_seeria_harjutus_id` (vt joonis 37), mida jällegi MDB andmebaasi korral ei tehta (vt joonis 36).



Joonis 36. Päring 4 täitmisplaan MDB korral



Joonis 37. Päring 4 täitmisplaan MDBI korral

MDB vs. MSV ja MSR

Päring 4 oli indeksiteta MonetDB andmebaasis 20 korda kiirem kui veerupõhises Microsoft SQL Server andmebaasis, kuid siiski kolm korda aeglasem kui reapõhises Microsoft SQL Server andmebaasis.

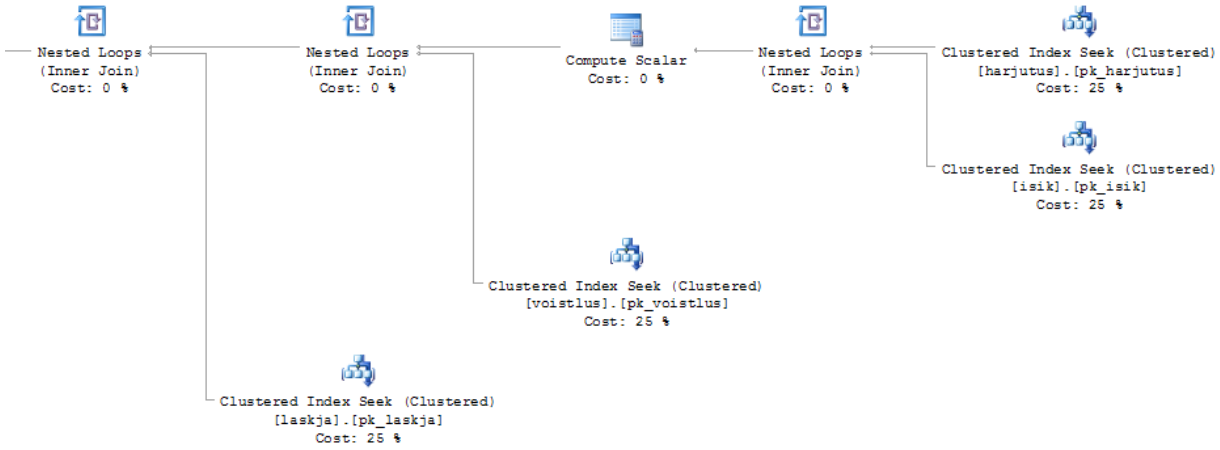
10.2.5 Päring 5

Leida harjutus *H*. Väljastada laskja klubi nimi, isikukood, ees- ja perenimi ühe sõnena, harjutuse identifikaator ning tüüp ja võistluse nimetus koos alguse ja lõpu ajaga.
Väljund: isik.isikukood + ' ' + isik.eesnimi + ' ' + isik.perenimi + ' ' + laskja.nimetus_klubi AS 'laskja', harjutus.nimetus_harjutus_tyyp AS 'harjutus', harjutus.harjutus_id, voistlus.nimetus, voistlus.alguse_kuupaev, voistlus.lopu_kuupaev
Predikaat: harjutus_id = *H*

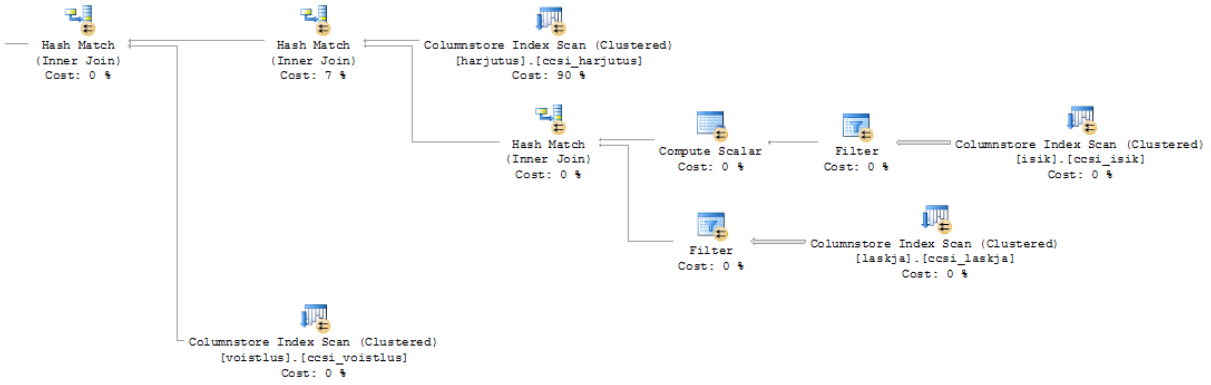
Test	**MSR** ***Külm***	**MSR** ***Soe***	**MSV** ***Külm***	**MSV** ***Soe***	**MDBI** ***Külm***	**MDBI** ***Soe***	**MDB** ***Külm***	**MDB** ***Soe***
Päring 5	0,2s	**0,001s**	0,6s	0,01s	1s	0,02s	0,7s	0,02s

MSR vs. MSV

Päring 5, mis on tüüpiline transaktsioonilistele operatiivandmete andmebaasidele, oli reapõhises Microsoft SQL Server andmebaasis oluliselt kiirem kui veerupõhises Microsoft SQL Server andmebaasis. Põhjus on täpselt sama, mis päring 4 puhul. Ka siin osutub optimaalsemaks reapõhise andmebaasi poolt kasutatav indeksi põhine otsing (*index seek* operatsioon), sest selles päringus on vaja üles leida ainult üks rida (olem) (vt joonis 38). Kuna veerupõhine indeks aga ei toeta indeksi põhist otsingut e *index seek* operatsiooni, siis MSV andmebaasis kulubki enamik päringu täitmise ajast (90%) (vt joonis 39) tabeli *harjutus* läbikäimiseks, et üles leida see üksik rida (olem). Ka siin on, reapõhise andmebaasi korral, vaja päringu täitmiseks lugeda üksnes 10 lehekülge, kuid veerupõhise jaoks üle 10000 lehekülje (vt tabel 12).



Joonis 38. Päring 5 täitmisplaan MSR korral



Joonis 39. Päring 5 täitmisplaan MSV korral

Tabel 12. Päring 5 IO statistika MSR ja MSV korral

Reapõhise salvestusega andmebaasi `IO` statistiline väljund

`Table 'harjutus'. Scan count 0, **logical reads 3**, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.`
 	
`Table 'laskja'. Scan count 0, **logical reads 2**, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.`
 	
`Table 'voistlus'. Scan count 0, **logical reads 2**, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.`
 	
`Table 'isik'. Scan count 0, **logical reads 2**, physical reads 0, read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads 0.`

Veerupõhise salvestusega andmebaasi `IO` statistiline väljund

```
Table 'harjutus'. Scan count 8, logical reads 10782, physical reads 0,
read-ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-
ahead reads 0.

Table 'voistlus'. Scan count 8, logical reads 0, physical reads 0, read-
ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead
reads 0.

Table 'isik'. Scan count 8, logical reads 0, physical reads 0, read-ahead
reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead reads
0.
Table 'laskja'. Scan count 8, logical reads 0, physical reads 0, read-
ahead reads 0, lob logical reads 0, lob physical reads 0, lob read-ahead
reads 0.
```

Huvitaval kombel on veerupõhise andmebaasi puhul tabelite *voistlus*, *isik* ja *laskja* juures *logical reads* 0. See võib tulla sellest, et need tabelid asuvad *deltastore*'s (ridade arv ei ületa lävendit) ja IO statistika ei näita *deltastore*'s loetud lehekülgede arvu.

MDBI vs. MDB

Mõlemates MonetDB andmebaasis toimus päringu 5 täitmine sama kiiresti.

MSR vs. MDB ja MDBI

Antud päring oli reapõhises Microsoft SQL Server andmebaasis kümme korda kiirem kui veerupõhises Microsoft SQL Server andmebaasis ning 20 korda kiirem kui kummaski MonetDB andmebaasis.

10.2.6 Ridade lisamine

Lisame harjutusele *H* uue seeria numbriga *N* summaga *S*.

Test	**MSR** ***Külm***	**MSR** ***Soe***	**MSV** ***Külm***	**MSV** ***Soe***	**MDBI** ***Külm***	**MDBI** ***Soe***	**MDB** ***Külm***	**MDB** ***Soe***
Lisamine	0,1s	**0,001s**	0,2s	**0,001s**	60s	14s	0,2s	0,05s

MSR vs. MSV

Mõlemad Microsoft SQL Server andmebaasid olid andmete sisestamise kiiruse poolest võrdsed. Veerupõhise salvestamisega andmebaasis toimus ridade lisamine sama kiiresti, sest

uute ridade lisamine toimub *deltastore*'i ja *deltastore* tegelikult kujutab endast tavalist reapõhist salvestamist.

MDBI vs. MDB

Indeksitega MonetDB andmebaasis oli rea lisamine oluliselt aeglasem kui ilma indeksiteta andmebaasis. Võrreldes mõlema andmebaasi `INSERT` lause täitmisplaane (vt joonis 40 ja 41) näeme, et MDBI oma on oluliselt keerulisem. MDBI täitmisplaanis on näha, et andmete sisestamiseks tehakse väga palju operatsioone, mis on seotud indeksite kasutamise ja uuendamisega.

```
+-----------------------------------------------------------------------------+
| rel                                                                         |
+=============================================================================+
| insert(                                                                     |
| | table(laskmine.seeria) [ seeria.harjutus_id NOT NULL, seeria.seeria_numbe |
: r NOT NULL, seeria.summa NOT NULL, seeria.%TID% NOT NULL ]                  :
| |  [ int[tinyint "1"] as L1.L1, smallint[tinyint "6"] as L2.L2, smallint[ti |
: nyint "100"] as L3.L3 ]                                                     :
| )                                                                           |
+-----------------------------------------------------------------------------+
```

Joonis 40. Rea lisamise täitmisplaan MDB korral

```
+-----------------------------------------------------------------------------+
| rel                                                                         |
+=============================================================================+
|  REF 1 (2)                                                                  |
|  [ int[tinyint "1"] as L4.L1, smallint[tinyint "6"] as L4.L2, smallint[tiny |
: int "100"] as L4.L3, sys.rotate_xor_hash(sys.hash(int[tinyint "1"] as L4.L1 :
: ), int "22", smallint[tinyint "6"] as L4.L2) as seeria.%pk_seeria ]         :
| insert(                                                                     |
| | & REF 1                                                                   |
| | insert(                                                                   |
| | | table(laskmine.seeria) [ seeria.harjutus_id NOT NULL HASHCOL , seeria.s |
: eeria_number NOT NULL, seeria.summa NOT NULL, seeria.%TID% NOT NULL, seeria :
: .%pk_seeria NOT NULL HASHIDX , seeria.%fk_seeria_harjutus_id NOT NULL JOINI :
: DX laskmine.seeria.fk_seeria_harjutus_id ]                                  :
| | | project (                                                               |
| | | | join (                                                                |
| | | | | table(laskmine.harjutus) [ harjutus.harjutus_id NOT NULL HASHCOL ,  |
: harjutus.voistlus_id NOT NULL, harjutus.nimetus_harjutus_tyyp NOT NULL, har :
: jutus.isikukood_isik_laskja NOT NULL, harjutus.%TID% NOT NULL, harjutus.%pk :
: _harjutus NOT NULL HASHIDX , harjutus.%uc_harjutus_komb_laskja_ja_voistlus_ :
: ja_harjutus_tyyp NOT NULL HASHIDX , harjutus.%fk_harjutus_nimetus_harjutus_ :
: tyyp NOT NULL JOINIDX laskmine.harjutus.fk_harjutus_nimetus_harjutus_tyyp,  :
: harjutus.%fk_harjutus_isikukood_isik_laskja NOT NULL JOINIDX laskmine.harju :
: tus.fk_harjutus_isikukood_isik_laskja, harjutus.%fk_harjutus_voistlus_id NO :
: T NULL JOINIDX laskmine.harjutus.fk_harjutus_voistlus_id, harjutus.%idx_har :
: jutus_nimetus_harjutus_tyyp NOT NULL HASHIDX , harjutus.%idx_harjutus_voist :
: lus_id NOT NULL HASHIDX  ] COUNT ,                                          :
| | | | | & REF 1                                                             |
| | | | ) [ harjutus.harjutus_id NOT NULL = L4.L1 ]                           |
| | | ) [ L4.L1, L4.L2, L4.L3, seeria.%pk_seeria, harjutus.%TID% NOT NULL as  |
: seeria.%fk_seeria_harjutus_id ]                                             :
| | )                                                                         |
| )                                                                           |
+-----------------------------------------------------------------------------+
```

Joonis 41. Rea lisamise täitmisplaan MDBI korral

MSR vs. MSV vs. MDB

Andmete sisestamine oli Microsoft SQL Server andmebaasides oluliselt kiirem kui ilma indeksiteta MonetDB andmebaasis. Põhjus on selles, et MonetDB on puhas veerupõhine andmebaasisüsteem aga MSR ja MSV andmebaasides toimub uute ridade lisamine kasutades reapõhist salvestamist.

10.2.7 Ridade uuendamine

Uuendame harjutuses *H* lastud seeria numbriga *N* väärtust asendades selle uue väärtusega *S*.

Test	**MSR *Külm***	**MSR *Soe***	**MSV *Külm***	**MSV *Soe***	**MDBI *Külm***	**MDBI *Soe***	**MDB *Külm***	**MDB *Soe***
Uuendamine	0,1s	**0,03s**	0,3s	**0,03s**	1,4s	0,1s	0,9s	0,08s

MSR vs. MSV vs. MDB vs. MDBI

Andmete uuendamise operatsiooni kiirused olid mõlemas Microsoft SQL Server andmebaasis jällegi võrdsed. Ilma indeksiteta MonetDB andmebaasis toimus uuendamine natukene kiiremini kui indeksitega MonetDB andmebaasis. Võrreldes omavahel Microsoft SQL Server ja MonetDB andmebaasisüsteeme, siis ridade uuendamine MDB andmebaasis oli ligi kolm korda aeglasem kui MSR või MSV andmebaasis.

10.2.8 Ridade kustutamine

Kustutame harjutuses *H* lasud seeria numbriga *N*.

Test	**MSR *Külm***	**MSR *Soe***	**MSV *Külm***	**MSV *Soe***	**MDBI *Külm***	**MDBI *Soe***	**MDB *Külm***	**MDB *Soe***
Kustutamine	0,1s	**0,05s**	0,2s	**0,05s**	24s	61s	23s	57s

MSR vs. MSV vs. MDB vs. MDBI

Mõlemas Microsoft SQL Server andmebaasis oli kustutamise operatsioon sama kiire. Samuti oli kustutamise operatsiooni kiirused võrdsed ka mõlemas MonetDB andmebaasis. Microsoft

SQL Serveris olevates andmebaasides toimus kustutamise operatsioon aga väga palju (üle 1200 korra) kiiremini kui kummaski MonetDB andemebaasisüsteemi andmebaasis.

MDB/MDBI (külm) vs. MDB/MDBI (soe)

Testides kustutamise operatsiooni kiirust andmebaasisüsteemis MonetDB, tuli lisaks välja ka selle ootamatu mõju kõikidele järgnevatele operatsioonidele (sh nii `SELECT`, `UPDATE` kui ka `DELETE`). See puudutab operatsioone, mis kasutavad seda tabelit, kust andmed kustutati. Teine kustutamine (soe) oli mõlema andmebaasi puhul ligi kolm korda aeglasem. Näiteks võtame vaatluse alla järgneva `SELECT` päringu.

```
SELECT * FROM seeria WHERE harjutus_id = 1;
```

Antud päringu täitmise ajaks on MDB andmebaasis umbes 0,007 sekundit. Järgnevalt kustutame ühe rea tabelist *seeria* lausega.

```
DELETE FROM seeria WHERE harjutus_id = 10000000 AND seeria_number = 6;
```

Käivitades uuesti sama `SELECT` päringu, saame sama päringu täitmise ajaks juba 26 sekundit. Teisisõnu, päringu kiirus muutub peale ülaltoodud `DELETE` lause käivitamist tuhandeid kordi aeglasemaks. Vaadates `SELECT` lause päringuplaani (`TRACE`), näeme et kõige suurem osa ajast kulub ühe BAT vahetabeli koostamiseks.

```
+----------+---------------------------------------------------------------------+
| ticks    | stmt                                                                |
+==========+=====================================================================+
|        0 | X_3 := sql.mvc();                                                   |
|       10 | X_7:bat[:oid,:int] =<tmp_407>[60000000] := sql.bind(X_3=0,"laskm |
:          : ine","seeria","harjutus_id",0);                                     :
| 26112159 | X_4:bat[:oid,:oid] =<tmpr_1326>[59999999] := sql.tid(X_3=0,"lask |
:          : mine","seeria");                                                    :

...

|        0 | end s4_1;                                                           |
| 26233856 | function user.s4_1(A0=1);                                           |
| 26233886 | X_5:void  := user.s4_1(1);                                          |
+----------+---------------------------------------------------------------------+
```

Tuleb välja, et MonetDB ei kustuta tabelist andmeid, vaid hoiab sisemiselt infot selle kohta, millised read on kustutatud. Seega moodustatakse tabelist *seeria* BAT vahetabel, kus on üks rida vähem (kustutatud rida) kui 60 miljonit, see aga võtab väga palju aega.

11. Järeldused

Antud peatükis tuuakse välja, kuidas pidasid paika töö alguses kirjeldatud hüpoteesid lähtudes praktilise töö osa tulemustest. Tuleb rõhutada, et hinnangud hüpoteeside kehtivusele põhinevad kahel andmebaasisüsteemil ja hulgal näiteoperatsioonidel. Igal süsteemil on omad realiseerimisest tulenevad eripärad. Seega ei väida see analüüs, et hüpoteeside kehtivus on kinnitatud või tagasi lükatud tehnoloogia (reapõhine salvestamine; veerupõhine salvestamine) korral üldiselt.

Veerupõhise andmete salvestamisega andmebaasisüsteemides töötavad kiiremini päringud, mis töötlevad palju ridu (10% ja rohkem tabeli ridu) aga kasutavad ja tagastavad vähe veerge.

Päring 1, mis kasutab tabelist *harjutus* ja tabelist *seeria* 10% ridadest ja 57% (seitsmest neli) nende tabelite veerge, oli veerupõhistes andmebaasides maksimaalselt (MDB) 10 korda kiirem kui reapõhises (MSR) ja seega loen antud hüpoteesi kehtivaks.

Veerupõhise andmete salvestamisega andmebaasisüsteemides päringu poolt tagastatav või kasutatav veergude hulk mõjutab nähtavalt päringu kiirust.

Antud hüpoteesi kontrollimiseks loodi päring 2, mis oli sarnane päringuga 1, kuid sisaldas `WHERE` klauslis liiaseid veerge, mistõttu lause kasutas 100% viidatud tabeli veergudest. Reapõhises andmebaasis (MSR) muutus päring 2 8%, veerupõhises andmebaasis (MSV) 20% ja veerupõhises andmebaasis (MDB) kolm korda aeglasemaks kui päring 1. Vaatamata sellele, et kõik andmebaasisüsteemid olid mõjutatud lisatud veergudest (tuleb arvestada ka loogika operatsioonide kulukust), mõjutas see veerupõhiseid andmebaase rohkem ja seega loen ka antud hüpoteesi kehtivaks.

Veerupõhise andmete salvestamisega andmebaasisüsteemidel ei tohiks olla erilist eelist, kui päringud töötlevad vähe ridu (1% ja alla kõikidest tabeli ridadest).

Antud hüpoteesi kontrollisin päringuga 3 ja 4, mis töötlesid vastavalt 0,1% ja 0,01% tabeli *seeria* ridadest. Päring 3 siiski loodetud tulemust ei andnud ja oli veerupõhises andmebaasis (MSV) ligi kümme korda kiirem kui reapõhises (MSR). Päring 4 aga juba oli reapõhises andmebaasis (MSR) kiirem kui üheski veerupõhises andmebaasis. Kuna antud hüpoteesis

eeldati, et vähe ridu on 1% ja vähem tabeli kõikidest ridadest, siis ei saa antud hüpoteesi kehtivaks lugeda.

Reapõhise andmete salvestamisega andmebaasisüsteemid peaksid olema oluliselt kiiremad, kui üles on vaja leida ainult üks rida (andmed ühe olemi kohta).

Päring 5 otsis tabelist *harjutus* üles ühe rea e andmed ühe olemi kohta. Antud päring oli reapõhises andmebaasis (MSR) 10 korda kiirem kui parimas veerupõhises andmebaasis (MSV). Sellest tulemusest lähtudes loen hüpoteesi kehtivaks.

Reapõhise andmete salvestamisega andmebaasisüsteemides toimub uute ridade lisamine kiiremini.

Kui võrrelda omavahel Microsoft SQL Serveris loodud andmebaase, siis võiks antud hüpoteesi lugeda mitte kehtivaks, sest seal olid ajad võrdsed. Arvestada tuleks aga ka, et MSV korral lisatakse uued read *deltastore*'i, mis on tegelikult kasutab reapõhist salvestamist. Sellest lähtudes peaks võrdlema pigem MSR ja MDB andmebaase. Võttes aluseks pigem nende kahe andmebaasi võrdluse, siis peab antud hüpotees pigem paika, sest rea lisamine MDB andmebaasis toimus oluliselt aeglasemini.

Reapõhise andmete salvestamisega andmebaasisüsteemides toimub ridade kustutamine kiiremini.

Andmete kustutamise kiirused olid Microsoft SQL Server andmebaasides jällegi võrdsed (põhjuseks *deltastore*). Seega ka siin tuleb kasutada MSR ja MDB andmebaaside võrdlust. Andmete kustutamine oli reapõhises andmebaasis (MSR) oluliselt kiirem kui veerupõhises andmebaasis (MDB) ja seega loen antud hüpoteesi kehtivaks.

Reapõhise andmete salvestamisega andmebaasisüsteemides toimub ridade muutmine kiiremini.

Andmete uuendamise kiirused olid Microsoft SQL Server andmebaasides jällegi võrdsed (põhjuseks *deltastore*). Seega ka selle hüpoteesi tõestamiseks või ümberlükkamiseks peaks kasutama andmebaaside MSR ja MDB võrdlus. MSR andmebaasis toimub ridade uuendamine mitu korda kiiremini kui MDB andmebaasis. Sellest tulemusest lähtudes loen antud hüpoteesi kehtivaks.

Veerupõhise salvestamisega andmebaasisüsteemides loodud andmebaasid peaksid andmemahult olema oluliselt väiksemad.

Microsoft SQL Server andmebaasisüsteemi korral oli andmebaasi andmemaht reapõhiselt salvestatult (ilma indeksiteta) ligi 1,4GB, kuid veerupõhiselt salvestatult kõigest 211MB. Andmebaaside andmemahtude vahe kasvas veelgi suuremaks, kuna reapõhisele andmebaasile tuli lisada ka täiendavad indeksid. Sellest tulemusest lähtudes loen antud hüpoteesi kehtivaks.

Antud töös leitud tulemustest lähtudes soovitan andmeaitade või teiste suurandmeid hoidvate andmebaaside korral kindlasti kaaluda veerupõhise salvestamisega andmebaasisüsteeme. Eriti kasulikus võib see osutuda siis, kui andmebaasides viiakse läbi analüütilisi päringuid e päringuid, mis peavad läbi vaatama palju ridu. Selliste päringute alla kuuluvad näiteks kokkuvõttefunktsioone `SUM` ja `COUNT` kasutavad päringud. Samas ei olnud veerupõhised andmebaasid väga aeglased ka üksiku olemi otsimisel ja seega võivad osutuda sobilikuks kui tehakse ka seda tüüpi otsinguid. Andmebaasisüsteemi MonetDB korral soovitan andmebaasis indekseid mitte luua. Indeksite lisamine kasvatas minu näite korral andmebaasi andmemahtu ligi kolm korda ja tegi peaaegu kõik päringud aeglasemaks. Kahjuks kehtib see ka primaarvõtme, välisvõtme ja unikaalsuse kitsenduste jaoks, sest need kehtestatakse sisemiselt läbi indeksite. Kitsenduste ära jätmine tekitab aga võimaluse andmebaasi lisada ärireeglitele mittevastavaid andmeid. Seega tuleks iga süsteemi korral eraldi testida, kas ja kui palju mõjutavad loodud kitsendused selles andmebaasis tehtavate päringute kiiruseid ja analüüsida võimalikke valede andmete mõju süsteemile ning selle alusel valida endale sobilik lahendus. Samuti tasuks MonetDB puhul vältida andmete kustutamist, kuna kustutamise operatsioon võib kaasa tuua olulist päringute kiiruste langust. Need MonetDB iseärasused ei ole kindlasti seotud mitte veerupõhise andmete salvestamise meetodiga, vaid pigem andmebaasisüsteemi enda realisatsiooniga.

12. Kokkuvõte

Reapõhine ja veerupõhine salvestamine on erinevad viisid, kuidas SQL-andmebaasisüsteemid võivad andmebaasi sisemisel tasemel andmeid esitada. Käesoleva töö eesmärk oli uurida, kuidas mõjutab veerupõhine ja reapõhine andmete salvestamine kahes SQL-andmebaasisüsteemis nende abil loodud andmebaasides toimuvate andmekäitluse operatsioonide jõudlust ning andmebaasi andmemahtu. Töö alguses püstitati kirjanduse põhjal üldised hüpoteesid rea- ja veerupõhise andmete salvestamise kohta, mille kehtivusele anti töö lõpuks konkreetsete süsteemide ja praktiliste katsete põhjal hinnang. Üks andmebaasisüsteemidest (MonetDB 5) oli puhas veerupõhise salvestamisega andmebaasisüsteem ning teine (Microsoft SQL Server 2014) oli reapõhise salvestamisega andmebaasisüsteem, kuid kus oli realiseeritud indeksi tüüp, mis salvestab andmed veerupõhiselt.

Eesmärgi saavutamiseks tutvuti kõigepealt teooriaga ning loodi selle paremaks mõistmiseks mõistekaarte (kasutades UMLi klassidiagramme). Otsiti teisi selles valdkonnas tehtud uuringuid ning toodi välja see, mis eristab uut planeeritavat uurinut eelnevatest. Seejärel loodi neli andmebaasi. Esimene neist loodi reapõhise salvestamisega Microsoft SQL Server andmebaasisüsteemis. Teine loodi samuti Microsoft SQL Server andmebaasisüsteemis, kuid seekord kasutati veerupõhist salvestamist läbi veerupõhise indeksi. Kolmas andmebaas loodi veerupõhise salvestusega MonetDB andmebaasisüsteemis. Neljas andmebaas loodi samuti MonetDB andmebaasisüsteemis, kuid sellele ei lisatud ühtegi täiendavat indeksit. Viimases andmebaasis ei loodud ka primaarvõtme, unikaalsuse ja välisvõtme kitsendusi, sest nende loomine viiks indeksite tekkimiseni. Loodud andmebaasides käivitati viis erinevat päringut ning üks ridade lisamise, uuendamise ja kustutamise lause ning mõõdeti nende kiiruseid. Päringutest üks oli tüüpiline tehingutöötluse süsteemi päring, mis otsib andmeid konkreetse olemi kohta ja ülejäänud koondandmete leidmiseks mõeldud päringud, mida vajavad üldjuhul analüüsi ning aruandluse süsteemid. Andmemuudatuste laused olid tüüpilised tehingutöötluse süsteemidele, sest nende sisuks oli üksikute ridade lisamine/muutmine/kustutamine.

Töö tulemusena leiti, et veerupõhise salvestamisega andmebaasid omavad suurt eelist päringute puhul, mille korral peab süsteem tulemuse saavutamiseks lugema suure osa tabeli andmetest (ridadest), kuid mitte kõiki veerge. Sellised on just koondandmete leidmise

päringud, mida kasutatakse tihti analüüsi ja aruandluse süsteemides. Samuti leiti, et andmebaasid mis salvestasid andmeid veerupõhiselt, olid andmemahu poolest oluliselt väiksemad. Töö lõpus kinnitati ka enamik kirjanduse alusel loodud hüpoteese.

Töö edasiarendamiseks saaks käsitletud andmebaasisüsteemide kõrval uurida ka laiaveerulisi (*wide-column*) NoSQL andmebaasisüsteeme nagu näiteks Apache Cassandra. Kuna sellist tüüpi andmebaasisüsteemide andmemudelis ei ole välisvõtmeid ning ülesehitatud andmestruktuuri põhjal ei saa teha *join* operatsioone, võib andmebaasisüsteemide objektiivne võrdlemine keerukas osutuda. (Cassandra data modeling, 2015) Samuti oleks huvitav võrrelda reapõhist ja veerupõhist salvestust „tunnel store“ andmete salvestamise mudeliga, mis väidetavalt erineb mõlemast eelnimetatust ja pakub väga head jõudlust ning mille realiseerib uue põlvkonna (NewSQL) SQL-andmebaasisüsteem *JustOneDB* (Core Innovation, 2015).

Lõpetuseks sooviksin kindlasti tänada töö juhendajat Erki Eessaart’t igakülgse abi eest.

13. Summary

Row storage and columnar storage are different ways how SQL Database Management Systems (DBMSs) could represents data at the internal database level. The goal of the thesis was to investigate how the use of row storage and columnar storage in two SQL DBMSs influences the speed of data management operations as well as the size of the databases created in those systems. At the beginning of the thesis a set of hypotheses about row storage and columnar storage was presented based on literature review. At the end of the thesis those hypotheses were evaluated based on the particular systems and practical experiments. One of the DBMSs was a DBMS (MonetDB 5) that offers purely columnar storage and the other one (Microsoft SQL Server 2014) was a DBMS that offers row storage but also supports index type that stores data in a columnar manner.

In order to achieve this goal, we firstly reviewed the literature and created concept maps (based on UML class diagrams) to better understand the theory. We also searched previous experiments in this field and described what are the differences of the planned experiment and the previous experiments. Next, four different databases were created. First database was created in Microsoft SQL Server by using row storage. Second database was also created in Microsoft SQL Server, but in this case column-store index was used to store data column wise. Third database was created in MonetDB, which offers columnar storage. Fourth database was also created in MonetDB, but in this case without any additional index as well as primary key, unique and foreign key constraints. The constraints were not created because these lead to the creation of indexes. Five different select queries as well as one insert, update, and delete operation were executed in all these databases and their speed was measured. From the select queries one query was typical for transactional databases by searching for single entity. Others were aggregate queries, which are more typical for analytical databases. Insert, update, and delete operations affected only a single row, which also makes them more typical for transactional systems.

A finding of this thesis was that databases with the columnar storage have great advantages over databases with row storage when it comes to queries that process a lot of table rows but only a subset of columns. These queries are usually aggregate queries, which are usually executed in analytical or reporting systems. Another significant finding was that databases

that use columnar storage use a lot less disk space compared to databases with same data but row storage. In the end of this thesis most of the hypothesis, which were formulated by the means of literature, were confirmed.

This research could be expanded by considering wide-column NoSQL DBMSs like Apache Cassandra. This kind of DBMSs do not support foreign keys and it is also impossible to do join operations, which means that it will be hard to compare this DBMS to others objectively. (Cassandra data modeling, 2015) It would also be interesting to compare row storage and column storage with the new storage type called „tunnel store“. Its inventors claim that it differs from the other two and offers very good performance. It is implemented by a new generation (NewSQL) DBMS *JustOneDB* (Core Innovation, 2015).

Finally I would like to express my gratitude to Erki Eessaar, supervisor of this work, for all the help he provided.

14. Kasutatud kirjandus

1. Abadi, J. D., Madden, R. S., Nachem, N. (2008). Column-Stores vs. Row-Stores: How Different Are They Really? – SIGMOD ’08, 967-980. [Online] http://dl.acm.org/citation.cfm?id=1376616.1376712 (28.04.2015)

2. Ailamaki, A., DeWitt, J. D., Hill, D. M. (2002). Data Page Layouts for Relational Databases on Deep Memory Hierarchies. – The VLDB Journal 11(3), 198-215. [Online] Springer (18.03.2015)

3. ATM: The Cache Machine! TMurgent. [WWW] http://www.tmurgent.com/Tools/ATM/Default.aspx (06.04.2015)

4. Boncz, P., Zukowski, M., Nes, N. (2005). MonetDB/X100: Hyper-Pipelining Query Execution. – Proceedings of the 2005 CIDR Confrence. [Online] http://db.cs.berkeley.edu/cs286/papers/monet-cidr2005.pdf (28.04.2015)

5. Clustered Index Structures. Microsoft Developer Network. [WWW] https://msdn.microsoft.com/en-us/library/ms177443.aspx (04.10.2014)

6. Columnstore Indexes Described. Microsoft Developer Network. [WWW] https://msdn.microsoft.com/en-us/library/gg492088%28v=sql.120%29.aspx (04.04.2014)

7. Computing. CERN. [WWW] http://home.web.cern.ch/about/computing (29.03.2015)

8. Data compression. MonetDB Documentation. [WWW] https://www.monetdb.org/Documentation/Guide/Compression (06.04.2015)

9. DB-Engines Ranking. DB-ENGINES. [WWW] http://db-engines.com/en/ranking (21.03.2015)

10. E-teadmik. Vallaste. [WWW] http://www.vallaste.ee (26.04.2015)

11. Halverson, A., Beckmann, L. J., Naughton, F. J., DeWitt, J. D. (2006). A Comparison of C-Store and Row-Store in a Common Framework. [Online] http://minds.wisconsin.edu/handle/1793/60514 (18.03.2015)

12. Harizopoulos, S., Liang, V., Abadi, J. D., Madden, S. (2006). Performance Tradeoffs in Read-Optimized Databases. – VLDB ’06 Proceedings, 487-498. [Online] http://dl.acm.org/citation.cfm?id=1182635.1164170 (18.03.2015)

13. Heap Structures. Microsoft TechNet. [WWW] https://technet.microsoft.com/en-us/library/ms188270%28v=sql.105%29.aspx (05.04.2014)

14. Ideros, S., Groffen, F., Nes, N., Manegold, S., Mullender, S., Kersten, M. (2012). MonetDB: Two Decades of Research in Column-oriented Database Architectures. [Online] http://stratos.seas.harvard.edu/files/MonetDebull2012.pdf (20.03.2015)

15. Jewell, D., Barros, D. R., Diederich, S., Duijvestijn, M. L., Hammersley, M. (2014). Performance and Capacity Implications for Big Data. [Online] http://www.redbooks.ibm.com/redpapers/pdfs/redp5070.pdf (29.03.2015)

16. Core Innovation. JustOne. [WWW] https://www.justonedb.com/products/justonedb/core-innovation/ (31.12.2015)

17. Lamb, A., Fuller, M., Varadarajan, R., Tran, N., Vandiver, B., Doshi, L., Bear, C. (2012). The Vertica Analytic Database: C-Store 7 Years Later. - Proceedings of the VLDB Endowment, Volume 5 Issue 12, 1790-1801. [Online] http://vldb.org/pvldb/vol5/p1790_andrewlamb_vldb2012.pdf (18.03.2015)

18. Larson, P., Clinciu, C., Hanson N. E., Oks, A., Price, L. S., Rangarajan, S., Surna, A., Zhou, Q. (2011). SQL Server Column Store Indexes. – SIGMOD ’11, 1177-1184. [Online] http://dl.acm.org/citation.cfm?id=1989448 (18.03.2015)

19. Liarou, E., Idreos, S., Manegold, S., Kersten, M. (2012) MonetDB/DataCell: Online Analytics in a Streaming Column-Store. – Proceedings of the VLDB Endowment, Volume 5 Issue 12, 1910-1913. [Online] http://dl.acm.org/citation.cfm?id=2367535 (28.04.2015)

20. Mis ja milleks on big data? DELFI LHV finantsportaal. [WWW] http://lhv.delfi.ee/news/4781213?locale=et (27.10.2015)

21. Norris, R. (2010). Data Challenges for Next-generation Radio Telescopes. [Online] http://arxiv.org/ftp/arxiv/papers/1101/1101.1355.pdf (29.03.2015)

22. Puustusmaa, S. (2012). Sisulise tähendusega võtmete ja surrogaatvõtmete kasutamise eelised ning puudused SQL-andmebaasides. Bakalaureusetöö. Tallinna Tehnikaülikooli Informaatikainstituut.

23. Query timing. MonetDB Documentation. [WWW] https://www.monetdb.org/Documentation/Cookbooks/SQLrecipes/QueryTiming (06.04.2015)

24. SPARC/DBMS Study Group. (1975). Interim report – SIGMOD Volume 7 Issue 2, 1975. [Online] http://portalparts.acm.org/990000/984332/fm/frontmatter.pdf (12.12.2015)

25. Stonebraker, M., Abadi, J. D., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madde, S., O'Neil, E., O'Neil, P., Rasin, A., Tran, N., Zdonik, S. (2005). C-Store: A Column-oriented DBMS. VLDB '05 Proceedings, 553-564. [Online] http://dl.acm.org/citation.cfm?id=1083658 (01.03.2015)

26. Table statistics. MonetDB Documentation. [WWW] https://www.monetdb.org/Documentation/Cookbooks/SQLrecipes/statistics (08.10.2015)

27. Understanding Pages and Extents. Microsoft TechNet. [WWW] https://technet.microsoft.com/en-us/library/ms190969%28v=sql.105%29.aspx (04.04.2014)

Lisad

Lisa 1

SQL laused tabelite loomiseks koos `CHECK` kitsendustega mõlema Microsoft SQL Server andmebaasisüsteemi andmebaasi jaoks.

```
CREATE TABLE laskja (
      isikukood_isik_laskja VARCHAR ( 11 ) NOT NULL,
      nimetus_klubi VARCHAR ( 30 ) NOT NULL,
      isikukood_isik_peatreener VARCHAR ( 11 ),
      CONSTRAINT cc_laskja_laskja_ei_tohi_olla_iseenda_peatreeneriks CHECK
(isikukood_isik_laskja <> isikukood_isik_peatreener)
);

CREATE TABLE harjutus_tyyp (
      nimetus VARCHAR ( 30 ) NOT NULL,
      seeriate_arv SMALLINT NOT NULL
      CONSTRAINT cc_harjutus_tyyp_seeriate_arv_suurem_kui_null CHECK
(seeriate_arv > 0)
);

CREATE TABLE voistlus (
      voistlus_id INTEGER NOT NULL,
      nimetus VARCHAR ( 100 ) NOT NULL,
      alguse_kuupaev DATE NOT NULL,
      lopu_kuupaev DATE NOT NULL,
      CONSTRAINT cc_voistlus_lopu_kuupaev_suurem_kui_alguse_kuupaev CHECK (
lopu_kuupaev >= alguse_kuupaev)
);

CREATE TABLE klubi (
      nimetus VARCHAR ( 30 ) NOT NULL
);

CREATE TABLE isik (
      isikukood VARCHAR ( 11 ) NOT NULL,
      eesnimi VARCHAR ( 30 ) NOT NULL,
      perenimi VARCHAR ( 30 ) NOT NULL
);

CREATE TABLE seeria (
      harjutus_id INTEGER NOT NULL,
      seeria_number SMALLINT NOT NULL
      CONSTRAINT cc_seeria_seeria_number_suurem_kui_null CHECK
(seeria_number > 0),
      summa SMALLINT DEFAULT 0 NOT NULL
      CONSTRAINT cc_seeria_summa_null_kuni_sada CHECK (summa >= 0 AND summa
<= 100)
);

CREATE TABLE harjutus (
      harjutus_id INTEGER NOT NULL,
      voistlus_id INTEGER NOT NULL,
      nimetus_harjutus_tyyp VARCHAR ( 30 ) NOT NULL,
      isikukood_isik_laskja VARCHAR ( 11 ) NOT NULL
);
```

Lisa 2

SQL laused tabelite loomiseks koos `CHECK` kitsendustega mõlema MonetDB andmebaasisüsteemi andmebaasi jaoks.

```
CREATE TABLE laskja (
      isikukood_isik_laskja VARCHAR ( 11 ) NOT NULL,
      nimetus_klubi VARCHAR ( 30 ) NOT NULL,
      isikukood_isik_peatreener VARCHAR ( 11 )
      CONSTRAINT cc_laskja_laskja_ei_tohi_olla_iseenda_peatreeneriks CHECK
(isikukood_isik_laskja <> isikukood_isik_peatreener)
);

CREATE TABLE harjutus_tyyp (
      nimetus VARCHAR ( 30 ) NOT NULL,
      seeriate_arv SMALLINT NOT NULL
      CONSTRAINT cc_harjutus_tyyp_seeriate_arv_suurem_kui_null CHECK
(seeriate_arv > 0)
);

CREATE TABLE voistlus (
      voistlus_id INTEGER NOT NULL,
      nimetus VARCHAR ( 100 ) NOT NULL,
      alguse_kuupaev DATE NOT NULL,
      lopu_kuupaev DATE NOT NULL
      CONSTRAINT cc_voistlus_lopu_kuupaev_suurem_kui_alguse_kuupaev CHECK (
lopu_kuupaev >= alguse_kuupaev)
);

CREATE TABLE klubi (
      nimetus VARCHAR ( 30 ) NOT NULL
);

CREATE TABLE isik (
      isikukood VARCHAR ( 11 ) NOT NULL,
      eesnimi VARCHAR ( 30 ) NOT NULL,
      perenimi VARCHAR ( 30 ) NOT NULL
);

CREATE TABLE seeria (
      harjutus_id INTEGER NOT NULL,
      seeria_number SMALLINT NOT NULL
      CONSTRAINT cc_seeria_seeria_number_suurem_kui_null CHECK
(seeria_number > 0),
      summa SMALLINT DEFAULT 0 NOT NULL
      CONSTRAINT cc_seeria_summa_null_kuni_sada CHECK (summa >= 0 AND summa
<= 100)
);

CREATE TABLE harjutus (
      harjutus_id INTEGER NOT NULL,
      voistlus_id INTEGER NOT NULL,
      nimetus_harjutus_tyyp VARCHAR ( 30 ) NOT NULL,
      isikukood_isik_laskja VARCHAR ( 11 ) NOT NULL
);
```

Lisa 3

SQL laused primaarvõtme, välisvõtme ja unikaalsuse kitsenduste loomiseks Microsoft SQL Server andmebaasisüsteemi reapõhise andmebaasi ja MonetDB andmebaasisüsteemi indeksitega andmebaasi jaoks.

```
ALTER TABLE laskja ADD CONSTRAINT pk_laskja PRIMARY KEY
(isikukood_isik_laskja);
ALTER TABLE harjutus_tyyp ADD CONSTRAINT pk_harjutus_tyyp PRIMARY KEY
(nimetus);
ALTER TABLE voistlus ADD CONSTRAINT pk_voistlus PRIMARY KEY (voistlus_id);
ALTER TABLE klubi ADD CONSTRAINT pk_klubi PRIMARY KEY (nimetus);
ALTER TABLE isik ADD CONSTRAINT pk_isik PRIMARY KEY (isikukood);
ALTER TABLE seeria ADD CONSTRAINT pk_seeria PRIMARY KEY (harjutus_id,
seeria_number);
ALTER TABLE harjutus ADD CONSTRAINT pk_harjutus PRIMARY KEY (harjutus_id);

ALTER TABLE voistlus ADD CONSTRAINT
uc_voistlus_komb_nimetus_ja_alguse_kuupaev UNIQUE (nimetus,
alguse_kuupaev);
ALTER TABLE harjutus ADD CONSTRAINT
uc_harjutus_komb_laskja_ja_voistlus_ja_harjutus_tyyp UNIQUE
(isikukood_isik_laskja, voistlus_id, nimetus_harjutus_tyyp);

ALTER TABLE seeria ADD CONSTRAINT fk_seeria_harjutus_id FOREIGN KEY
(harjutus_id) REFERENCES harjutus (harjutus_id)  ON DELETE CASCADE ON
UPDATE NO ACTION;
ALTER TABLE harjutus ADD CONSTRAINT fk_harjutus_nimetus_harjutus_tyyp
FOREIGN KEY (nimetus_harjutus_tyyp) REFERENCES harjutus_tyyp (nimetus)  ON
DELETE NO ACTION ON UPDATE CASCADE;
ALTER TABLE harjutus ADD CONSTRAINT fk_harjutus_isikukood_isik_laskja
FOREIGN KEY (isikukood_isik_laskja) REFERENCES laskja
(isikukood_isik_laskja)  ON DELETE NO ACTION ON UPDATE CASCADE;
ALTER TABLE harjutus ADD CONSTRAINT fk_harjutus_voistlus_id FOREIGN KEY
(voistlus_id) REFERENCES voistlus (voistlus_id)  ON DELETE NO ACTION ON
UPDATE NO ACTION;
ALTER TABLE laskja ADD CONSTRAINT fk_laskja_isikukood_isik_laskja FOREIGN
KEY (isikukood_isik_laskja) REFERENCES isik (isikukood)  ON DELETE NO
ACTION ON UPDATE NO ACTION;
ALTER TABLE laskja ADD CONSTRAINT fk_laskja_isikukood_isik_peatreener
FOREIGN KEY (isikukood_isik_peatreener) REFERENCES isik (isikukood)  ON
DELETE NO ACTION ON UPDATE NO ACTION;
ALTER TABLE laskja ADD CONSTRAINT fk_laskja_nimetus_klubi FOREIGN KEY
(nimetus_klubi) REFERENCES klubi (nimetus)  ON DELETE NO ACTION ON UPDATE
CASCADE;

CREATE INDEX idx_laskja_isikukood_isik_peatreener ON laskja
(isikukood_isik_peatreener );
CREATE INDEX idx_laskja_nimetus_klubi ON laskja (nimetus_klubi );
CREATE INDEX idx_harjutus_nimetus_harjutus_tyyp ON harjutus
(nimetus_harjutus_tyyp );
CREATE INDEX idx_harjutus_voistlus_id ON harjutus (voistlus_id );
```

Lisa 4

SQL laused klasterdatud veerupõhiste indeksite loomiseks Microsoft SQL Server andmebaasisüsteemi veerupõhist salvestamist kasutava andmebaasi jaoks.

```
CREATE CLUSTERED COLUMNSTORE INDEX ccsi_laskja ON laskja;
CREATE CLUSTERED COLUMNSTORE INDEX ccsi_harjutus_tyyp ON harjutus_tyyp;
CREATE CLUSTERED COLUMNSTORE INDEX ccsi_voistlus ON voistlus;
CREATE CLUSTERED COLUMNSTORE INDEX ccsi_klubi ON klubi;
CREATE CLUSTERED COLUMNSTORE INDEX ccsi_isik ON isik;
CREATE CLUSTERED COLUMNSTORE INDEX ccsi_seeria ON seeria;
CREATE CLUSTERED COLUMNSTORE INDEX ccsi_harjutus ON harjutus;
```

Lisa 5

```
package ee.siimpuustusmaa.main;

public class AndmeteGenereerija {

  private static final String FAILI_EESLIIDE = "laused";

  public static void main(String[] args) {
    TestandmeteGeneraator testandmeteGeneraator = new TestandmeteGeneraator();
    testandmeteGeneraator.setBulkMode(true);
    testandmeteGeneraator.genereeri(FAILI_EESLIIDE);
    System.out.println("Valmis");
  }
}

package ee.siimpuustusmaa.main;

import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.file.Files;
import java.nio.file.InvalidPathException;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;
import java.sql.Date;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Random;
import java.util.Set;
import java.util.concurrent.TimeUnit;

import org.apache.commons.lang3.RandomStringUtils;
import org.apache.commons.lang3.StringUtils;

public class TestandmeteGeneraator {

  private static final int HARJUTUSE_TYYP = 10;
  private static final int LASKJA = 1000;
  private static final int TREENER_JA_KLUBI = 10;
  private static final int VOISTLUS = 1000;
  private static final String VOISTLUSED_ALGUS = "2000-01-01";
  private static final int VOISTLUSED_KESTVUS_PAEVADES = 2;
  private static final int VOISTLUSED_VAHE_PAEVADES = 14;

  private static final int ISIKUKOOD_PIKKUS = 11;
  private static final int SEERIATE_ARV = 6;
  private static final int[] EESNIMI_MIN_MAX_PIKKUS = { 3, 10 };
  private static final int[] PERENIMI_MIN_MAX_ARV_PIKKUS = { 3, 12 };
  private static final int[] KLUBI_NIMETUS_MIN_MAX_PIKKUS = { 5, 12 };
  private static final int[] HARJUTUS_TYYP_NIMETUS_MIN_MAX_PIKKUS = { 10, 25 };
  private static final int[] VOISTLUS_NIMETUS_MIN_MAX_PIKKUS = { 8, 55 };
  private static final int[] SEERIA_SUMMA_MIN_MAX_ARV = { 50, 100 };

  private Random random = new Random();
  private boolean bulkMode = false;

  /**
   * 1. Luuakse unikaalsed isikukoodid. LASKJA + TREENER_JA_KLUBI tükki
   * 2. Luuakse isikud kasutades järjest eelnevas sammus loodud isikukoode
   * 3. Luuakse unikaalse nimetusega klubid. TREENER_JA_KLUBI tükki
   * 4. Luuakse laskjad kasutades järjest punktis 1 loodud isikukoode. LASKJA tükki.
   * 5. Luuakse unikaalse nimetusega harjutuse tüübid. HARJUTUSE_TYYP tükki
   * 6. Luuakse unikaalse nimetusega võistlused alates kuupäevast
   * VOISTLUSED_ALGUS, kestvusega VOISTLUSED_KESTVUS_PAEVADES päeva ning seda
   * iga VOISTLUSED_VAHE_PAEVADES päeva tagant. VOISTLUS tükki.
   * 7. Luuakse harjutused selliselt, et iga võistlusel iga harjutuses osaleb iga laskja
   * ehk kokku VOISTLUS * HARJUTUSE_TYYP * LASKJA tükki.
   * 8. Luuakse iga harjutuse kohta SEERIATE_ARV seeriat.
   *
   * @param failiEesliide
   */
  public void genereeri(String failiEesliide) {
    List<String> laused = new ArrayList<String>();
```

```
    // ISIK
    System.out.println("Loon isikuid...");
    List<String> isikukoodid = unikaalsedIsikukoodid(LASKJA + TREENER_JA_KLUBI);
    for (String isikukood : isikukoodid) {
      String lause = looInsertLause("isik", new String[]{"isikukood", "eesnimi", "perenimi"},
isikukood,
          juhuslikTahtedeJada(EESNIMI_MIN_MAX_PIKKUS[0], EESNIMI_MIN_MAX_PIKKUS[1]),
          juhuslikTahtedeJada(PERENIMI_MIN_MAX_ARV_PIKKUS[0], PERENIMI_MIN_MAX_ARV_PIKKUS[1])
          );
      laused.add(lause);
    }
    kirjutaKoikFaili(laused, failiEesliide, "isik");
    laused.clear();

    // KLUBI
    System.out.println("Loon klubisid...");
    List<String> klubid = unikaalsedNimetused(TREENER_JA_KLUBI,
KLUBI_NIMETUS_MIN_MAX_PIKKUS[0], KLUBI_NIMETUS_MIN_MAX_PIKKUS[1]);
    for (String klubi : klubid) {
      String lause = looInsertLause("klubi", new String[]{"nimetus"}, klubi);
      laused.add(lause);
    }
    kirjutaKoikFaili(laused, failiEesliide, "klubi");
    laused.clear();

    // LASKJA
    System.out.println("Loon laskjaid...");
    for (int i = 0; i < LASKJA; i++) {
      String lause = looInsertLause("laskja", new String[]{"isikukood_isik_laskja",
"nimetus_klubi", "isikukood_isik_peatreener"},
          isikukoodid.get(i), klubid.get(i%TREENER_JA_KLUBI), isikukoodid.get(LASKJA +
random.nextInt(TREENER_JA_KLUBI)));
      laused.add(lause);
    }
    kirjutaKoikFaili(laused, failiEesliide, "laskja");
    laused.clear();

    // HARJUTUS TYYP
    System.out.println("Loon harjutuse tüüpe...");
    List<String> harjutuseTyybid = unikaalsedNimetused(HARJUTUSE_TYYP,
HARJUTUS_TYYP_NIMETUS_MIN_MAX_PIKKUS[0], HARJUTUS_TYYP_NIMETUS_MIN_MAX_PIKKUS[1]);
    for (int i = 0; i < HARJUTUSE_TYYP; i++) {
      String lause = looInsertLause("harjutus_tyyp", new String[]{"nimetus", "seeriate_arv"},
          harjutuseTyybid.get(i), SEERIATE_ARV);
      laused.add(lause);
    }
    kirjutaKoikFaili(laused, failiEesliide, "harjutus_tyyp");
    laused.clear();

    // VÕISTLUS
    System.out.println("Loon võistlusi...");
    List<String> voistlused = unikaalsedNimetused(VOISTLUS,
VOISTLUS_NIMETUS_MIN_MAX_PIKKUS[0], VOISTLUS_NIMETUS_MIN_MAX_PIKKUS[1]);
    long hetk = Date.valueOf(VOISTLUSED_ALGUS).getTime();
    for (int id = 1; id <= VOISTLUS; id++) {
      String lause = looInsertLause("voistlus", new String[]{"voistlus_id", "nimetus",
"alguse_kuupaev", "lopu_kuupaev"},
          id, voistlused.get(id-1), new Date(hetk), new Date(hetk +
TimeUnit.DAYS.toMillis(VOISTLUSED_KESTVUS_PAEVADES)));
      laused.add(lause);
      hetk += TimeUnit.DAYS.toMillis(VOISTLUSED_VAHE_PAEVADES);
    }
    kirjutaKoikFaili(laused, failiEesliide, "voistlus");
    laused.clear();

    // HARJUTUS
    System.out.println("Loon harjutusi...");
    Path harjutuseFail = looUusFail(failiEesliide + "_" + "harjutus" + ".sql");
    int id = 1;
    for (int voistlusId = 1; voistlusId <= VOISTLUS; voistlusId++) {
      for (int tyypIndeks = 0; tyypIndeks < HARJUTUSE_TYYP; tyypIndeks++) {
        for (int laskjaIndeks = 0; laskjaIndeks < LASKJA; laskjaIndeks++) {
          String lause = looInsertLause("harjutus", new String[]{"harjutus_id", "voistlus_id",
"nimetus_harjutus_tyyp", "isikukood_isik_laskja"},
              id++, voistlusId, harjutuseTyybid.get(tyypIndeks),
isikukoodid.get(laskjaIndeks));
```

```
          laused.add(lause);
          kirjutaFailiHulkLauseid(laused, harjutuseFail, 10000);
        }
      }
    }

    kirjutaFaili(laused, harjutuseFail);
    laused.clear();

    // SEERIA
    System.out.println("Loon seeriaid...");
    Path seeriaFail = looUusFail(failiEesliide + "_" + "seeria" + ".sql");
    int harjutusteArv = VOISTLUS * HARJUTUSE_TYYP * LASKJA;
    for (int harjutusId = 1; harjutusId <= harjutusteArv; harjutusId++) {
      for (int seeriaNumber = 1; seeriaNumber <= SEERIATE_ARV; seeriaNumber++) {
        String lause = looInsertLause("seeria", new String[]{"harjutus_id", "seeria_number",
"summa"},
            harjutusId, seeriaNumber, (random.nextInt(SEERIA_SUMMA_MIN_MAX_ARV[1] -
SEERIA_SUMMA_MIN_MAX_ARV[0] + 1) + SEERIA_SUMMA_MIN_MAX_ARV[0]));
        laused.add(lause);
        kirjutaFailiHulkLauseid(laused, seeriaFail, 10000);
      }
    }
    kirjutaFaili(laused, seeriaFail);
    laused.clear();
  }

  private String looInsertLause(String tabel, String[] valjad, Object... vaartused) {
    if (bulkMode) {
      return StringUtils.join(vaartused, ',');
    } else {
      StringBuilder lause = new StringBuilder();
      lause.append("INSERT INTO "+tabel+" ("+StringUtils.join(valjad, ',')+") VALUES (");
      for (int i = 0; i < vaartused.length; i++) {
        if(vaartused[i] instanceof String) {
          lause.append("'"+vaartused[i]+"'");
        } else {
          lause.append(vaartused[i]);
        }

        if(i < vaartused.length-1) {
          lause.append(",");
        }
      }
      lause.append(");");
      return lause.toString();
    }
  }

  private void kirjutaFailiHulkLauseid(List<String> laused, Path fail, int hulk) {
    if (laused.size() > hulk) {
      kirjutaFaili(laused, fail);
      laused.clear();
    }
  }

  private void kirjutaKoikFaili(List<String> laused, String eesliide, String fail) {
    Path path = looUusFail(eesliide + "_" + fail + ".sql");
    kirjutaFaili(laused, path);
  }

  private void kirjutaFaili(List<String> laused, Path fail) {
    try {
      Files.write(fail, laused, Charset.defaultCharset(), StandardOpenOption.WRITE,
StandardOpenOption.APPEND);
    } catch (IOException e) {
      System.out.println("Väljundfaili kirjutamine ebaõnnestus");
      System.exit(-1);
    }
  }

  private Path looUusFail(String uueFailiNimi) {
    Path uusFail = null;
    try {
      uusFail = Paths.get(uueFailiNimi);
    } catch (InvalidPathException e) {
      System.out.println("Vigane väljundfaili asukoht!");
```

```
      System.exit(-1);
    }

    try {
      Files.deleteIfExists(uusFail);
      Files.createFile(uusFail);
    } catch (IOException e) {
      System.out.println("Väljundfaili loomine ebaõnnestus");
      System.exit(-1);
    }

    return uusFail;

  }

  private String juhuslikTahtedeJada(int minPikkus, int maxPikkus) {
    return RandomStringUtils.randomAlphabetic(random.nextInt(maxPikkus+1 - minPikkus) +
minPikkus);
  }

  private List<String> unikaalsedNimetused(int genereeritavHulk, int minPikkus, int maxPikkus)
{
    Set<String> unikaalsed = new HashSet<String>();
    while (unikaalsed.size() != genereeritavHulk) {
      unikaalsed.add(juhuslikTahtedeJada(minPikkus, minPikkus));
    }
    List<String> list = new ArrayList<String>();
    list.addAll(unikaalsed);
    return list;
  }

  private List<String> unikaalsedIsikukoodid(int arv) {
    Set<String> isikukoodid = new HashSet<String>();
    while (isikukoodid.size() != arv) {
      isikukoodid.add(RandomStringUtils.random(1, '3',
'4').concat(RandomStringUtils.randomNumeric(ISIKUKOOD_PIKKUS - 1)));
    }
    List<String> list = new ArrayList<String>();
    list.addAll(isikukoodid);
    return list;
  }

  public void setBulkMode(boolean bulkMode) {
    this.bulkMode = bulkMode;
  }

}
```

Lisa 6

Andmete lisamine SQL laused *bulk* meetodil Microsoft SQL Server andmebaasisüsteemis.

```
BULK INSERT isik
FROM 'C:\Users\Siim\Dropbox\siim\kooli_asjad\loputoo\testandmete
genereerimine\loputoo\laused_isik.sql'
WITH ( FIELDTERMINATOR = ',' );

BULK INSERT klubi
FROM 'C:\Users\Siim\Dropbox\siim\kooli_asjad\loputoo\testandmete
genereerimine\loputoo\laused_klubi.sql'
WITH ( FIELDTERMINATOR = ',' );

BULK INSERT laskja
FROM 'C:\Users\Siim\Dropbox\siim\kooli_asjad\loputoo\testandmete
genereerimine\loputoo\laused_laskja.sql'
WITH ( FIELDTERMINATOR = ',' );

BULK INSERT harjutus_tyyp
FROM 'C:\Users\Siim\Dropbox\siim\kooli_asjad\loputoo\testandmete
genereerimine\loputoo\laused_harjutus_tyyp.sql'
WITH ( FIELDTERMINATOR = ',' );

BULK INSERT voistlus
FROM 'C:\Users\Siim\Dropbox\siim\kooli_asjad\loputoo\testandmete
genereerimine\loputoo\laused_voistlus.sql'
WITH ( FIELDTERMINATOR = ',' );

BULK INSERT harjutus
FROM 'C:\Users\Siim\Dropbox\siim\kooli_asjad\loputoo\testandmete
genereerimine\loputoo\laused_harjutus.sql'
WITH ( FIELDTERMINATOR = ',' );

BULK INSERT seeria
FROM 'C:\Users\Siim\Dropbox\siim\kooli_asjad\loputoo\testandmete
genereerimine\loputoo\laused_seeria.sql'
WITH ( FIELDTERMINATOR = ',' );
```

Lisa 7

Andmete lisamine SQL laused *bulk* meetodil MonetDB andmebaasisüsteemis.

```
COPY INTO isik
FROM 'C:\Users\Siim\Dropbox\siim\kooli_asjad\loputoo\testandmete
genereerimine\loputoo\laused_isik.sql'
USING DELIMITERS ',','\n';

COPY INTO klubi
FROM 'C:\Users\Siim\Dropbox\siim\kooli_asjad\loputoo\testandmete
genereerimine\loputoo\laused_klubi.sql'
USING DELIMITERS ',','\n';

COPY INTO laskja
FROM 'C:\Users\Siim\Dropbox\siim\kooli_asjad\loputoo\testandmete
genereerimine\loputoo\laused_laskja.sql'
USING DELIMITERS ',','\n';

COPY INTO harjutus_tyyp
FROM 'C:\Users\Siim\Dropbox\siim\kooli_asjad\loputoo\testandmete
genereerimine\loputoo\laused_harjutus_tyyp.sql'
USING DELIMITERS ',','\n';

COPY INTO voistlus
FROM 'C:\Users\Siim\Dropbox\siim\kooli_asjad\loputoo\testandmete
genereerimine\loputoo\laused_voistlus.sql'
USING DELIMITERS ',','\n';

COPY INTO harjutus
FROM 'C:\Users\Siim\Dropbox\siim\kooli_asjad\loputoo\testandmete
genereerimine\loputoo\laused_harjutus.sql'
USING DELIMITERS ',','\n';

COPY INTO seeria
FROM 'C:\Users\Siim\Dropbox\siim\kooli_asjad\loputoo\testandmete
genereerimine\loputoo\laused_seeria.sql'
USING DELIMITERS ',','\n';
```